VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Gramatická evoluce v jazyce nezávislém na výpočetní platformě

Grammatical Evolution Based on a Platform Independent Language

2014 Tomáš Machala

VŠB - Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Diploma Thesis Assignment

Student: **Bc. Tomáš Machala**

Study Programme: N2647 Information and Communication Technology

Study Branch: 2612T025 Computer Science and Technology

Title: Gramatická evoluce v jazyce nezávislém na výpočetní platformě
Grammatical Evolution Based on a Platform Independent Language

Description:

The main aim is to create program of grammatical evolution that will be independent on the computer hardware. A part of program shall be suitable example like data fitting, artificial neural network fitting, etc. Proposed diploma thesis are within domain of bio-inspired computation, namely to evolutionary algorithms and symbolic regression.

1. Introduction into state of art of grammatical evolution.
2. Create program of grammatical evolution.
3. Test of grammatical evolution on selected examples.
4. Create step-by-step user manual.
5. Conclusion.

References:

[1] Koza J.R. 1998, Genetic Programming, MIT Press, ISBN 0-262-11189-6, 1998
[2] Koza J.R.,Bennet F.H., Andre D., Keane M. 1999, Genetic Programming III, Morgan Kaufnamm pub., ISBN 1-55860-543-6, 1999
[3] Lampinen Jouni, Zelinka, Ivan, New Ideas in Optimization & Mechanical Engineering Design Optimization by Differential Evolution. Volume 1. London: McGraw-Hill, 1999. 20 p. ISBN 007-709506-5
[4] Kvasnička V., Pospíchal J., Tiňo P., Evolučné algoritmy, STU Bratislava, ISBN 85-246-2000, 2000
[5] Zelinka I.: Analytic Programming by Means of Soma Algorithm. ICICIS'02, First International Conference on Intelligent Computing and Information Systems, Egypt, Cairo, 2002
[6] Zelinka Ivan, Evoluční výpočetní techniky - principy a alikace, BEN, Praha, 2008

Extent and terms of a thesis are specified in directions for its elaboration that are opened to the public on the web sites of the faculty.

Supervisor: **prof. Ing. Ivan Zelinka, Ph.D.**

Date of issue: 01.09.2013

Date of submission: 07.05.2014

doc. Dr. Ing. Eduard Sojka
Head of Department

prof. RNDr. Václav Snášel, CSc.
Dean of Faculty

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.*

V Ostravě 7. května 2014 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2014 .....

Rád bych na tomto místě poděkoval prof. Ing. Ivanu Zelinkovi, Ph.D. za vedení mé diplomové práce.

Poděkování

Tato práce byla vypracována s podporou projektu Rozvoj lidských zdrojů ve výzkumu a vývoji moderních soft computingových metod a jejich praktického využití, reg. č. CZ.1.07/2.3.00/20.0072 podpořeného Operačním programem Vzdělávání pro konkurenceschopnost, financovaného ze strukturálních fondů EU a státního rozpočtu ČR.

Abstrakt

Předmětem této diplomové práce je implementace gramatické evoluce v jazyce nezávislém na platformě. Gramatická evoluce je pokročilá optimalizační technika, která spadá do oblasti evolučních algoritmů. Možnosti jejího využití jsou široké a pokrývají spoustu odvětví inženýrských, ekonomických a dalších oborů. Práce je rozdělena do tří částí. První část je teoretická a pojednává o principech gramatické evoluci a evolučních algoritmů obecně. Druhá část popisuje mé implementace programů, které jsem v rámci této diplomové práce vytvořil a experimenty, které jsem s nimi prováděl. Poslední část obsahuje shrnutí této práce a nabízí možnosti dalšího vývoje.

Klíčová slova: Evoluční algoritmy, gramatická evoluce, diferenciální evoluce, SOMA, optimalizace, bioinspirované výpočty, symbolická regrese, deterministický chaos, logistická mapa

Abstract

The thesis aims on implementation of grammar evolution in a platform-independent language. Grammar evolution is an advanced optimization technique from a field of evolutionary algorithms. Its range of applications is very wide and covers many engineering, economical and other fields of science. The thesis is divided into three parts. The first one is rather theoretical and describes the principles of grammar evolution and evolutionary algorithms in general. The second part describes my own implementations of grammar evolution and the experiments for which I have used them. The last part concludes achieved results and also proposes the ways of further research of the topic.

Keywords: Evolutionary algorithms, grammar evolution, differential evolution, SOMA, optimization, bio-inspired computing, symbolic regression, deterministic chaos, logistic map

List of acronyms

RNG	–	Random Number Generator
PRNG	–	Pseudo-Random Number Generator
GE	–	Grammar Evolution
DE	–	Differential evolution
SOMA	–	Self-Organizing Migrating Agents
API	–	Application Programming Interface
BNF	–	Backus-Naur Form
CR	–	Cross Rate
GDE	–	Grammatical Differential Evolution
GSOMA	–	Grammatical Self-Organizing Migrating Agents
IV	–	Initialization Vector
CLI	–	Command Line Interface
MSIL	–	Microsoft Intermediate Language
CFG	–	Context-Free Grammar
GP	–	Genetic Programming
PSO	–	Particle Swarm Optimization
KISS	–	Keep It Simple Stupid
DRY	–	Don't Repeat Yourself
SoC	–	Separation of Concerns
GPU	–	Graphics Processing Unit

Contents

List of Tables

List of Figures

List of source code listings

1 Introduction

The need of problem optimization is very old and covers various disciplines including engineering, economics, physics, biology and many others. In fact, many real-world problems can be defined as an optimization problem. The goal of a typical optimization is to maximize productivity or performance of some process or device or to minimize waste. Many more or less sophisticated optimization techniques have been developed over time. While the simplest problems involving functions of a single variable may be solved using basic math, many real-world problems require more complex tools.

For a long time in history optimization methods were based on still more and more complicated methods usually involving exact mathematics. However, with increasing complexity of problems to be optimized a need for more powerful and flexible optimization techniques arisen. In mid-sixties, evolutionary algorithms were developed to address these demands. They are considered a powerful tool with many advantages over traditional optimization techniques.

One of the biggest advantages of evolutionary algorithms is that unlike many other traditional optimization techniques, they don't depend upon mathematical models of problems. Actually, the only precondition of using an evolutionary algorithm is the ability to evaluate a concrete candidate solution. In other words, the only thing that matter is whether or not it is possible to evaluate a solution once it is presented.

The thesis focuses on grammar evolution which is an advanced evolutionary technique suitable for symbolic regression. Classical grammar evolution uses genetic algorithms as a computational core that manipulates genetic information. Later parts of the thesis deals with an idea of replacing the genetic-based core by DE or SOMA. There has been made an experiment that measures performance of these alternative versions compared to the classical GE.

Another part of the thesis aims on the topic of random number generators and theirs relation to evolutionary algorithms, mainly to GE. There is a theoretical part regarding RNGs. Another experiment that has been done as a part of the thesis aims to measure feasibility of a PRNG based on logistic map with GE, GDE and GSOMA.

The last parts of the thesis conclude these experiments and propose possible ways of further research on the matter [1] [2] [3].

2 Evolutionary Techniques

Evolutionary Techniques are numerical algorithms inspired by the principles of Darwin's and Mendel's evolution theory. From the most general perspective, evolutionary techniques are heuristic algorithms that can be further categorized as deterministic or stochastic. Deterministic techniques, as implied by their names, behave always the same. It is possible to predict the next step in any phase of their execution and given the same initial state, their behavior is replicable. Stochastic techniques, on the other hand, incorporate a factor of randomness into their execution and so may give different results when executed multiple times, even for the same initial states. If we ignore the fact that various PRNGs would actually also behave deterministically, given the same initialization vectors.

However, this is not the only way of categorizing various evolutionary techniques. Another approach is to categorize them based on whether they use a concept of population, as seen in nature. Some of them, such as DE or GE do, while another such as SOMA use another approach where there is still the same set of individuals who just migrate over the state space [2] [3] [4] [5].

2.1 Evolution Workflow

According to the Darwin's and Mendel's evolution theory, various organisms evolve in a way that descendant individuals are spawned by their parents who then eventually clear out the life space for the new generation. Individuals are subjects of mutation which is a process that alters their function by randomly modifying parts of their genetic information. Such phenomenon can also be seen in nature. The evolution works in cycles during which individuals unsuitable for their environment extinct and the suitable individuals prevail. That leads to a continuous improvement of whole population [2] [3]. Various evolutionary techniques usually follow a process similar to that shown on figure 1. Individual steps of the process are further explained in following chapters.

1. Definition of Evolution Parameters

   The evolution is run according to a set of control parameters. Individual evolutionary algorithms require different set of them, as discussed in later parts of the thesis. A cost function is also defined in this phase. Cost function is a mathematical model of the given problem that is to be optimized. Its purpose is to evaluate concrete individuals by how well they solve given problem in a way that their goodness is mutually comparable. We can thing of the cost function as of an environment in which the individuals live.

   An evolution may run indefinitely (or until stopped) but it is also possible to define a terminating condition that stops the evolution once satisfied. Different evolutionary techniques use different terminating conditions, as described in later parts of the thesis. Terminating conditions also belong to the set of control parameters and thus are also to be defined in this initial phase [3].

2. Creation of an Initial Population

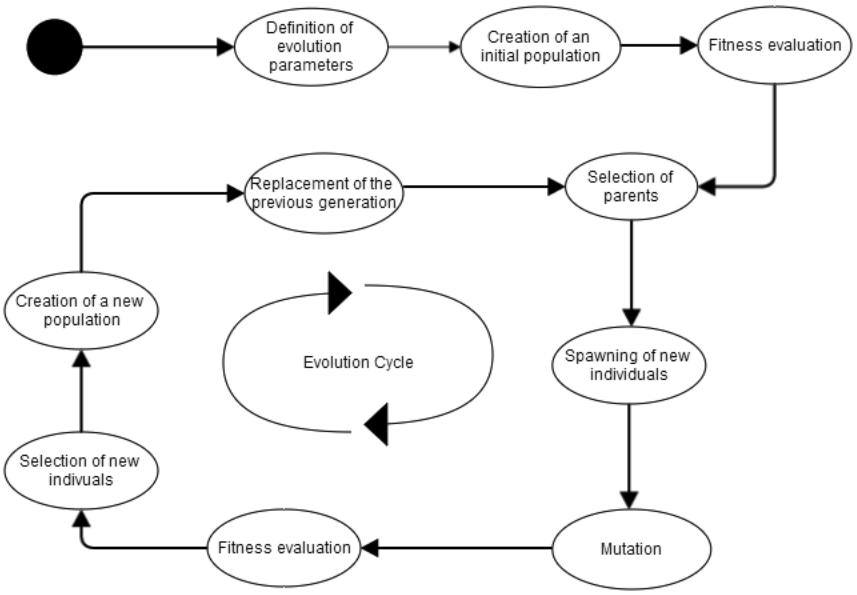


Figure 1: Evolution Cycle

In the next phase, an initial population of individuals is created. Since evolutionary techniques requires a previous generation in order to evolve its descendant generation which is not available yet, the initial population have to be created by generating random genetic information. That consequently produces completely random set of individuals. However not every genetic information represents a valid individual. The fitness function may have various constraints that restrict the individual's attributes in some way. Hence it may be needed to repeat the process more times than is the desired population size [2] [3].

3. Fitness Evaluation

   In this phase, all individuals are assigned a fitness value. The fitness is a number (typically a real one, in most circumstances) that expresses the suitability of the individual for solving the problem defined by a fitness function. Since it is needed to call the cost function to determine whether an individual is valid or not when creating an initial population, the line between this one and the previous step is a bit blurred from implementation point of view but conceptually, we can think of it as shown in the figure 1 [3].

4. Selection of Parents

   As soon as individuals are evaluated, the next step is to choose a subset of them that will be allowed to reproduce and spawn a new generation. This is the moment when natural selection takes place. Badly performing individuals (i.e. those with bad fitness values) are pushed to extinct while more suitable individuals (those

with better fitness values) preserve and are given a chance to spawn even better descendants by combining parts of their generic information (so called crossing). Exact details of how the crossing is done depend on the evolutionary technique used. E.g. roulette-wheel, tournament or rank selection is commonly used with GE while DE, e.g., uses different approach. All three crossing algorithms mentioned will be discussed later in the thesis [3].

5. Spawning of New Individuals

   The parents that were selected in the previous step get involved in creation of a new generation. Again, the exact details differs depending on the crossing algorithm used but in in most cases an individual is created by combining various parts of genetic information of two or four parents. Although the four-parent variant is not commonly seen in nature, is used by DE where it gives good results [3].

6. Mutation

   Mutation is a natural phenomenon that randomly alters small parts of genetic information which consequently affects some attributes of an individual. However, since computer is a digital device with extremely small probability that some of bits that represents the genetic information would swap, it has to be implemented artificially. The purpose of mutation is the same both in nature and in computational applications – it prevents the population from stagnating and also helps to discover new attributes that are not present among any of individual from the population. Although the mutation is an important part of an evolution, it is a random process that may improve an individual as well as it can degrade or totally disable it [3].

7. Fitness Evaluation

   Same as fitness values of all parents were evaluated in step 3, now it is time to evaluate the spawned individual that will make a new population. Again, as in step 3, from the implementation point of view, this step is not clearly separated from the previous. The fitness function may have some hard constraints defined that cause that some individuals represents totally unacceptable solutions and are rejected. Every individual to extinct leaves an empty spot that need to be filled by another individual [3].

8. Fitness Evaluation

   Once all individuals from the population mutates, they are evaluated by a cost function that assigns them a cost value. The cost value may be further converted to a fitness value by a fitness function. The difference between the two will also be discussed later [3].

9. Selection of New Individuals

   Same as a subset of the randomly generated population has been selected to involve in spawning of a new generation, their descendants are now about to do the same [3].

10. Replacement of the Previous Generation

    In this final step that finished a single evolution iteration (so called generation or migration cycle, depending on the evolutionary algorithm used) old population is replaced by a new one. This usually involves rewriting of the old population in memory unless some implementation keeps the old generations for some reason. If a terminating condition is set, it is evaluated in this step and the evolution eventually stops, if satisfied. If the condition is not satisfied or is not defined at all, the evolution continues by the step 4 as indicated on the figure 1 [3].

2.2 Basic Concepts

This chapter aims to describe various terms and concepts that are used through the rest of the thesis. There is also a figure 2 that helps to understand how various terms relate to each other.

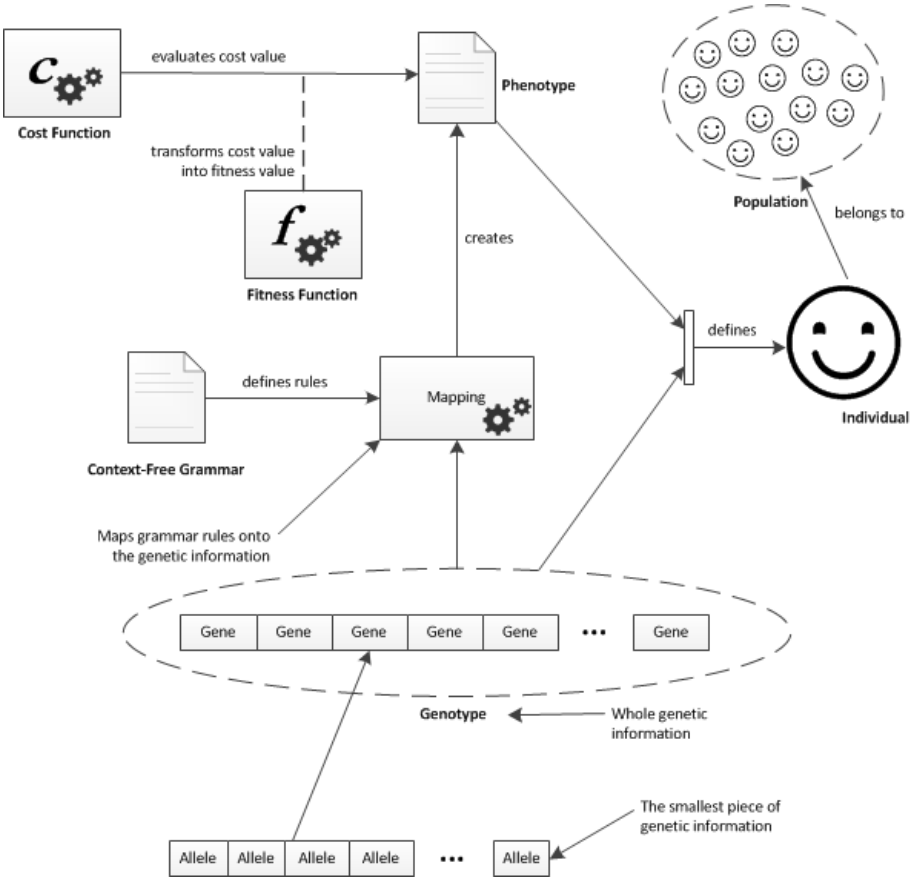


Figure 2: Basic Concepts Relation

Genotype

Genotype represents the whole genetic information of single individual. Genetic information is expressed by a chromosome which is a vector of genes. As for computational applications, we can think of chromosome and genotype as equivalent terms [2] [3].

Gene

Gene is a single part of genotype. The whole vector of genes represents a genotype. A gene is usually represented by a numerical data type in computational applications. However, it is technically possible to represent it also by non-numerical types such as text strings, enumerations or constants or even a mixture of these types. Non-numerical types are used mainly with algorithms that are out of scope of this thesis, such as analytical programming and so will not be further discussed. For the rest of this thesis a gene is assumed to be a number [2] [3].

Allele

Allele is the smallest piece of genetic information. In computational applications the smallest piece of information is represented by a single bit. Although genes are internally defined by series of bits, we usually do not work on such low level. A single gene is more commonly the smallest part that we work with [2] [3].

Phenotype

The phenotype represents the external appearance of an individual that had grown up from certain genetic information. In biological sense, it represents all attributes of an organism that carries genetic information, such as a plant or animal. In computational applications, phenotype is the candidate solution [2] [3].

Individual

An individual represents a single candidate solution of the problem that the evolution optimizes. A group of individuals that were evolved during a single iteration make a population. With algorithms such as DE the individual is represented by its genetic information (genotype) and a fitness value. However, in case of symbolic regression algorithms such as GE, an individual consists of one extra attribute – a phenotype. In nature, phenotype represents the external appearance of an individual. If it was a rabbit, for instance, we could see that it is furry, white, with two eyes, one nose etc. That is its external appearance – the phenotype. The phenotype is defined by the genetic information carried by genotype. In computational applications, phenotype is the expression that was produced by mapping CFG rules onto a genotype [2] [3].

Generation

Many of evolutionary algorithms work in cycles during which they produce a constant number of individuals that all together makes a generation. Basically, the term is similar to a generation as we think of it in nature. There are however some differences. In nature, for example, the number of spawned descendants varies and they do not spawn iteratively, all at once. There are some algorithms that also use a concept of generations but in a slightly different form. With SOMA, for instance, single evolution iteration is referred to as a migration cycle instead of generation. The reason is the individuals do not extinct when a new migration cycle is executed. There is still the same set of individuals who migrate. They persist through the whole evolution process [2] [3].

Population

The population is simply a set of individuals that belong to the same generation or that were migrated during the same migration cycle in case of SOMA [2] [3].

Cost Value

Cost Value is a numerical attribute of an individual that represent how suitable it is for its environment. In other words, the cost value expresses how well solution of the problem that the evolution optimizes the individual represents. Depending on the cost function's design, either lower or higher numerical value represents a better solution. In the second case, the value should be converted by a fitness function to a fitness value which is essentially a cost value transformed in a way that lower numbers represent better solutions [2] [3].

Cost Function

Cost function is an essential component of an evolution. Its purpose is to evaluate candidate solutions by a cost value. It is the only part that cannot be componentized and reused. Each problem requires its own cost function. The quality of a fitness function is critical for evolution success. If the cost function is implemented incorrectly, the whole evolution may converge to a non-optimal solution, i.e. solution that is optimal according to the defective cost function but does not reflect the actual real-world problem correctly.

Depending on the problem optimized, in may not be possible to implement the cost function as a self-contained software component. It may rely on an input from an external system. It is even possible (and meaningful, in some cases) to build a cost function that requires a human operator as a mediator who manually sets an external system in a way requested by the evolution, measures some phenomenon and inputs the measured data back into the cost function [2] [3].

With real-world problems, valid ranges of concrete individuals' attributes are usually restricted in some way. Depending on the concrete problem, restrictions may arise from various physical or economical limitations. It is, e.g., not possible to build a material of

negative thickness or the proposed solution might be too expensive to carry out. There are two counter-measures that a cost function can implement in order to deal with such inappropriate candidate solutions:

- **Soft constraints** that consider unacceptable solutions valid but penalizes their cost values so that they become handicapped and thus are more likely to extinct.
- **Hard constraints** that just reject defect solution so that they are immediately removed from a population.

Both soft and hard constraints make sense in certain cases. It is not possible to say which one is better. With soft constraints, the evolution might actually end up with a defect solution. On the other hand, if hard constraints were used and were too restrictive the evolution could be rather a random process trying to find any solution that is not defect at least. Soft constraint may give useful information about the direction that the evolution should take [2] [3].

Fitness Value

Fitness value is a cost value transformed in a way that lower values represents better solutions that higher values, as mentioned in the previous paragraph [2] [3].

Fitness Function

The purpose of a fitness function is to convert cost value to a fitness value. Although it would look good if zero was the ideal value, in some cases it may not be possible to define the ideal solution. Then it is enough to multiply the cost value by -1. If the cost function is defined in a way that it already evaluates better solutions by lower numbers, no fitness transformation is needed at all. However, even in such cases the fitness function can be used to convert values range. I.e., if it is known that a cost function return values from -3745.81 to 7041.55 the fitness function may convert such values so that they fit into a range from 0 to 100 which looks better from users' perspective [2] [3].

Terminating Condition

Terminating condition causes the evolution to stop once satisfied. It is usually evaluated at the end of each iteration. Depending on the purpose that the terminating condition follows it can be defined for example (but not limited to) by:

- The fitness value of the best individual;
- The difference in fitness values between the best and the worst individual;
- The number of generations;
- The time limit.

The purpose of the best fitness variant is obvious – once an optimal or nearly optimal solution is found, it makes no sense to keep trying. Similarly, the success can be defined by a number of generations instead of achieved result. DE for instance, uses the second option mentioned. It aims to stop the evolution when it has converged to certain solution and is unlikely to find a better one. For applications where the available time is restricted (user-interactive applications, i.e.), the condition can also be represent a time limit [2] [3].

2.3 Evolutionary Algorithms Performance

Performance of an evolutionary algorithm is obviously a metric that express how fast an evolution runs in some testing environment. It may seem natural to compare performance of different evolutions by the number of generations that they are capable of evolving within a given time span. However, this is not the right approach for the following reasons:

- Although it is technically possible to measure performance of GE, GDE and GSOMA that way, there are also evolutionary algorithms that don't use the concept of generations. So this approach is applicable only to a certain subset of evolutionary algorithms.
- More importantly, the number of evolved generations is not a valid metric of evolution progress in terms of finding a solution of a given problem. Given the same population sizes, different algorithms needs to evolve highly different number of generations to achieve comparable results. For example, both GE and GDE commonly need to evolve much more generations that GSOMA (called migration cycles it that case) to give a comparably good solutions.

The valid way of mutually comparing performance of various evolutionary algorithms is by counting how many times the fitness function have to be evaluated before an algorithm manages to find a solution of given quality. This can be visualized by a chart where there's the number of fitness function evaluations on the horizontal axis and the best fitness at the vertical axis. Of course, due to the random nature of evolutionary algorithms, it is important to repeat each test several times and compare average results [3].

One of the reasons why this is the right approach is because for most real problems, the evaluation of a fitness function is by far the most resource-consuming part of the evolution, especially if it depends on input from an external system.

3 Grammar Evolution

Grammar evolution is a variant of genetic programming based on a context-free grammar [6]. While the original GP by Koza [4] is tightly coupled with LISP, GE can be implemented in various languages and produce expressions according to grammar rules.

Most evolutionary algorithms (DE for instance) outputs a fixed set of parameter values that can be used in a function that is known in advance. Such algorithms are hence suitable for optimization of problems where it is enough to supply numerical values of their predefined set of parameters. GE works differently. Although it also internally works with numerical vectors, it doesn't output them directly. Instead, vector values are mapped onto rules defined by a context-free grammar which produces an expression. The expression may be anything that can be described by a CFG. Depending on the optimized problem, the CFG rules may lead to production of a mathematical or logical expression or even to a multiline program code in some programming language. Consequently, cost functions do not evaluate vectors of numbers as in the case of DE but rather phenotypes. Hence, GE can be used for symbolic regression, which makes it far more powerful tool than most of other evolutionary techniques.

The classic GE is tightly coupled with genetic algorithms that manipulate genetic information. I have made an implementation that follows this approach as a part of the thesis. However, since genetic algorithms fall short in terms of performance when compared to many other evolutionary algorithms, the part of this thesis deals with an idea of replacing the genetic part of a GE by DE or SOMA. Neither DE nor SOMA is capable of producing an expression in the way that GE do. However, the same as GE enhances genetic algorithms by that capability, an extra layer can be added on top of both DE and SOMA to enable that features. The grammar-enabled versions of these algorithms are referred to as GDE and GSOMA [5].

3.1 Selection

Selection is a process of choosing individuals whose genetic information will be involved in creation of a new population. In general, the evolution tries to preserve individuals that represent good solutions of the problem. In nature it means those individuals that are suitable for the environment that they live in. Consequently, individuals representing bad solutions are pushed to extinct. The pressure that discriminated bad solutions over good ones is referred to as selection strength.

The selection is not to be done in a way that the candidates were ordered by their fitness values and only a fixed number of the best would reproduce. Such process would lead to a problem called early convergence. The evolution would probably progress rapidly in several first generations but then it would stall in a local extreme and could not escape from it. Individuals with bad fitness values are also important in the evolution process. Although they are not particularly useful as candidate solutions, they may carry some good parts of the genetic information that good individuals are missing. If such bad individuals were totally eliminated, so were their genes and consequently, dominant

individuals lose the opportunity to get these good parts of genetic information and to overcome the local extreme.

Several selection algorithms are commonly used with GE. The following chapters describe three most commonly used techniques. All of them are available in the GE implementation that is a part of this thesis [3].

Tournament Selection

The first algorithm imitates a tournament, as indicated by its name. Tournament Selection incorporates both deterministic and non-deterministic approaches:

- Given number of individuals are picked from the whole population. They're chosen randomly. Given a perfect RNG being used, each individual has the same chance to be selected, disregarding its fitness value.
- These randomly chosen individuals are placed into an arena to conduct a fictional tournament.
- The tournament is the deterministic part of the algorithm. The winner is determined by its fitness value compared to others.

The number of individuals that are chosen to conduct a tournament is a parameter of Tournament Selection, referred to as tournament size. It has an impact on selection pressure. The higher tournament size, the stronger selection pressure there is. The usual tournament size is around 5 individuals [2] [3].

Depending on implementation, multiple clones of the same individual may or may not be placed into the arena. Theoretically, it would be more natural not to allow multiple clones of the same individual into the arena. On the other side, given usual tournament sizes vs. population sizes, the probability of a single individual being placed into the arena more than once is small. By ignoring these cases, we can omit some pieces of code from implementation, which may result in very small performance boost. Although the operation cost is trivial, whole evolution consists of trivial steps like these, executed many times.

Roulette-Wheel Selection

Another commonly used algorithm imitates a roulette-wheel. Individuals get assigned certain sectors on the roulette-wheel, based on their fitness value, so that better individuals occupy more space than worse ones. However, no individual is left-out at all. Sectors are distributed in a way that no empty sectors are left and all individuals together occupy full $360°$ circle, without overlapping. A spin is then made and an individual is selected. The roulette-wheel model is of course just theoretical. In practice, the algorithm is implemented differently, using just a line instead of wheel:

- Sum of all fitness values is computed.

- For each individual, the ratio of its fitness to the total sum is computed.
- Those ratios are put on a line next to each other, preserving references to individuals.
- A random number between zero and the line length is generated and used to determine the "winner".

This selection algorithm, unlike the previous one, does not have any control parameter [2] [3].

Rank Selection

Rank selection is similar to roulette-wheel selection. Its analogy could also be a roulette-wheel. The difference is that individuals are ordered by their fitness values ascending (the best one first) and assigned ordinal numbers from 1 on. The rest of the algorithm is the same as described in the previous chapter, with the only difference that those ordinal numbers are used to distribute individuals over the line instead of their original fitness values.

Since there is much less divergence in ordinal numbers that it would be in fitness values in most cases, rank selection causes lower selection pressure that the other algorithm. With roulette-wheel selection, e.g. an individual with 1000 times higher fitness value than its competitor would be 1000 times less likely to get selected. With rank selection, its chances would be much higher (unless the population is enormously large). In general, rank selection may be useful when the evolution seems to converge too fast with other selection algorithms. On the other side, it may not be suitable for simple problems where more aggressive algorithms would allow faster progress [2] [3].

Elitism

Dominant individuals are generally more likely to transfer their good genetic information into new generations. However, since randomness is involved it evolution process, they are not guaranteed not to extinct. When it happens, the evolution might end-up with worse solution than it had available among individuals of past generations. Obviously, that would be an unwanted outcome.

That is why the elitism is usually implemented. Its purpose is to copy the certain amount of best-performing individuals into a new population to override the risk of their extinction. The rest of the population is filled-up normally, by means of crossing and mutation.

The elitism level (or elitism strength) is a control parameter. Technically, it can be any number from zero to the population size minus one but mostly just one individual is used. It is enough to fulfill the purpose (i.e. to prevent losing the best solution) and larger numbers might cause an early convergence by increasing the selection pressure [2] [3].

3.2 Crossing

Crossing is a process of creation of a new individual by combining genes of its parents. Same as with selection, this can be achieved using several algorithms.

The first method is to randomly determine a point in both parents' genotypes and create a new individuals by taking genetic information from beginning up to the crossing point from the first parent and the rest (from the crossing point on) from the second parent (see fig. 3).

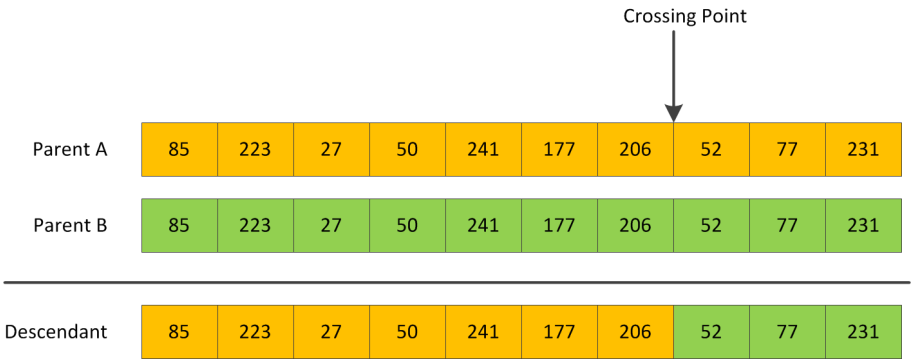


Figure 3: Single-point crossing

Another method randomly chooses two crossing points. A new individual is then made by taking the parts of the genetic information before the first and after the second crossing point, combining them with the part between crossing points from the second parent. Although this method may be appropriate in some cases, it is not used in the thesis because it makes a little sense with grammar evolution. With GE, an individual is represented by phenotype that has been created by mapping genotype onto a CFG. A change anywhere in the middle of the genotype totally changes the derivation tree and produces moderately different expression. Hence, even if just a single gene somewhere in the middle of a genotype would be changed, the derivation tree would be totally different from that point on as seen on figure 4 [2] [3].

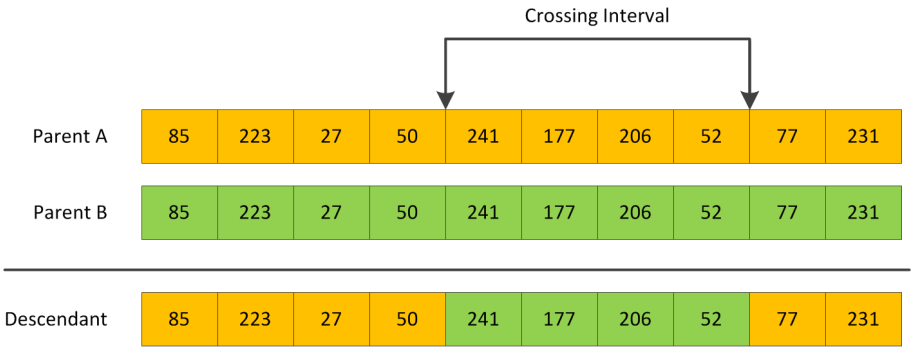


Figure 4: Interval crossing

3.3 Mutation

Mutation is a random process that affects genotypes of newly created individuals, which takes places after crossing. By affecting particular individual's genotype, mutation may have either positive or negative outcome. On the one side, an individual might get a feature that it cannot inherit from its parents since neither of them have such feature. On the other side, mutation may affect good parts of the genotype, thus degrading the individual. As a consequence, mutation also mitigates the risk of stagnation.

In nature, mutation takes place as a result of "transfer errors". In computational applications, it has to be implemented artificially, since computers are digital devices. Although data corruption causing a bit to spontaneously flip to the opposite state may occur even with digital devices as a result of background radiation or cosmic rays, it is not usable in any practical way.

The mutation strength is one of the control parameters. It is expressed by a number between 0 and 1 where 0 means that no mutation shall occur at all while 1 means that 100% of genetic information shall be altered. Mutation strength between 0.005 and 0.01 is recommended (altering from 0.5% to 0.1% of genetic information). Since individuals are represented by phenotypes created by mapping genotype onto a CFG with GE, even a small change anywhere in the middle of the genotype can totally changes the derivation tree and produce very different expression. The same happens with crossing. Hence, higher mutation strength is rather contra productive. The evolution process would be more random than evolutional.

It is possible to implement the evolution in a way that the mutation strength raises as the population converges to a common point. Similar approach is seen in nature where descendants produced by parents with similar genetic information are likely to be affected by a stronger mutation than their counterparts produced by more divergent parents. Same as the first case, the aim is to mitigate the risk of stagnation [7].

3.4 Genotype – Phenotype Mapping

As mentioned earlier, genetic information is represented as a vector of numbers by most evolutionary techniques, i.e. by DE, genetic algorithms or SOMA. The fact that GE outputs expressions instead of numerical vectors is made possible by a process that maps such vectors on rules defined by a context-free grammar.

Formally, a context-free grammar is defined by a tuple $G = \{N, T, P, S\}$ where:

- N is a finite set of non-terminal symbols. A non-terminal symbol is a symbol that has to be further rewritten by terminal numbers before the expression building is finished.
- T is a finite set of terminal symbols. A terminal symbol terminates a node of derivation tree, is indicated by its name.
- S is the initial symbol. As for BNF, the non-terminal defied on a first line is considered initial.

- P is a set of rewrite rules [3].

The BNF is a commonly used format to describe CFGs and it is also used by the programs included with the thesis.

Mapping Algorithm

The mapping process builds a derivation tree by applying rewrite rules determined by a gene value. Please consider the following sample vector (a genotype) and the grammar described by BNF:

```
G = (207, 130, 52, 246, 11, 1, 71, 195, 35)

<code>  ::= <line>
         |   <code> <line>
<line>  ::= <if-st>
         |   <op>
<if-st> ::= if(food_ahead()){<line>} else{<line>}
<op>    ::= left()
         |   right()
         |   move()
```

The derivation starts with the initial rule (which is the first one defined, in case of BNF). Since it has two possible rewrite rules that the algorithm can choose from, a first gene value is read to determine which rule to use. Since gene values ranges from 0 to 255 (if represented by a single byte) but there is rarely 255 rewrite options per one rule, the index is determined by the following formula:

$$I = G \bmod R \tag{1}$$

where G is the gene value and R is the number of rewrite rules that we can choose from. In this case, 207 mod 2 equals to 1 and so the "<code> <line>" option is chosen [3]. After this step, the derivation tree looks like figure 5:

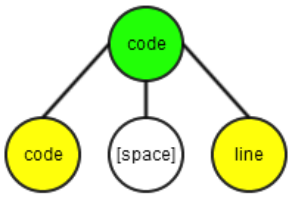


Figure 5: Derivation tree after the first step

There is at least one non-terminal symbol in the tree so the derivation isn't complete yet. Since the left-most derivation is used by convention, the code non-terminal is the

next to be rewritten. Another gene value is read to determine which rule to use etc., until all non-terminals are rewritten to terminals. The grammar and the genotype as specified eventually produce the derivation tree shown on figure 6.

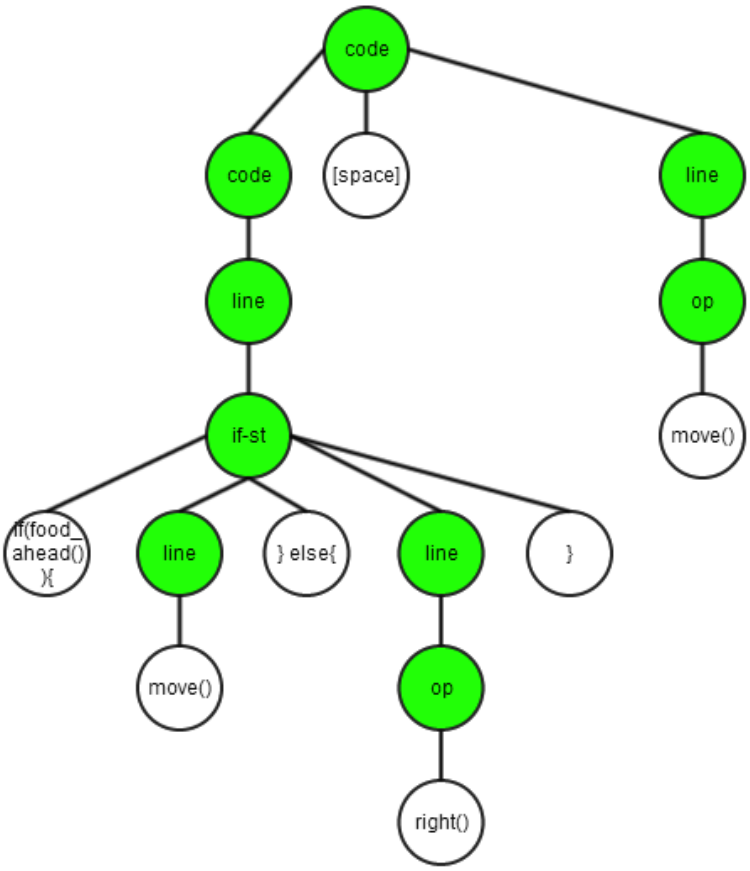


Figure 6: Final derivation tree

Finally, there is a string that this particular derivation tree represents (formatted for readability):

```
if (food_ahead()) {
    move()
} else {
    right()
}
move()
```

Dealing with Short Genotypes

There are cases when a mapping algorithm may exhaust all genes of a genotype before a derivation of an expression is finished. It can happen e.g. in following situations:

- When the derivation tree is lengthy and the genotype is just too short for it.
- If the grammar rules allows infinite derivation.

The mapping is an operation that is called very frequently during evolution's execution and so these situations happen regularly. There are two possible ways how to programmatically deal with them:

1. The genotype can be iterated in an infinite loop. If all genes are read, we start all over from the first gene. However, if we choose this approach, there should be some loop count limit. Otherwise, the evolution would stall and eventually run out of memory in case of derivation trees of infinite length. If the loop count limit is reached, the genotype is considered to be defect and is thrown away. If a genotype is thrown away a new one has to be evolved as a replacement. Otherwise, we would lower the number of individuals in population.

2. The genotype can be rejected right away, without even trying to iterate the gene more than once. This approach is a bit easier to implement but the first method is more appropriate. If the genotype length is set too low and most of the genotypes get thrown away, the overall performance of the evolution will suffer. This approach is hence less durable to incorrectly specified chromosome length.

Neither of these solutions is optimal. An appropriate genotype length should be set by the evolution's control parameter. However, mapping leads to construction of various derivation trees whose gene consumption may differ heavily. Thus it is not practical to use excessively lengthy genotypes just so that the evolution would not run out of it. The genotype length has a direct and easily observable impact on evolution's performance. Although various numerical operations with genes are inexpensive, they are invoked extremely frequently [2] [3].

4 Differential Evolution

Differential evolution is very popular evolutionary technique. It was developed by Ken Price and Rainer Storm in 1995. The thesis does not cover DE history. Please refer to [8] in a case of interest. Following chapters describe its control parameters and principles.

4.1 Control Parameters

Parameter	Range	Description
CR	$\langle 0; 1\rangle$ ($\langle 0.8; 0.9\rangle$ is recommended)	Cross Rate. Defines the probability that two individuals would cross. It has very similar meaning to the cross rate used with genetic algorithms. The value of 1 means that the individuals will always cross, while 0 means that the individual is not affected by the mutation and represents a copy of the original individual. Neither 0 nor 1 is likely to be the right value in most cases.
D	≥ 1, depending on problem definition	Problem dimensions. Represents the number of unknown variables of the problem being optimized. Hence, it is only affected when the problem is changed or redefined.
NP	Technically any number higher or equal than 4. Usually $\langle 10D; 100D\rangle$ where D is the number of dimensions.	The population size. There is no exact procedure to determine the right value. It is mainly a matter of experience with the concrete problem and with DE itself.
F	$\langle 0; 2\rangle$ ($\langle 0.3; 0.9\rangle$ is recommended)	Mutation constant. It affects how much an individual is affected before crossing, as described later.
Generation	> 0	Defines the number of generations to evolve [8].

Table 1: Description of control parameters

4.2 Population

DE uses the same concept as most of the other evolutionary techniques, namely GE those principled have been described earlier in the thesis. The explanation can be found there [8].

4.3 Mutation

What on the other hand differs from what was described in chapters dedicated to GE is mutation. The first difference is that it takes place before crossing, not after it. Another difference not only between DE and GE but also between replication processes in nature is that four parents are involved in spawning an offspring instead of two.

For each individual of the population another three different individuals referred to as r_1, r_2 and r_3 are chosen. A noise vector v is created according to the formula below. The difference between first two individuals is multiplied by the mutation constant F and the result is added to the third individual [8].

$$v_j = x^G_{r3,j} + F(x^G_{r1,j} - x^G_{r2,j}) \tag{2}$$

4.4 Crossing

Crossing process uses the noisy vector described in previous chapter. The vector of the fourth individual that had not been used until now is combined with the noisy vector to produce another individual referred to as trial vector. The combination process generates a random number for each vector dimension. If the generated number is lower than the CR parameter, a value of the noisy vector is used. Otherwise, the fourth individual's vector component is used. The trial vector then competes with the fourth individual as described in the next chapter [8].

4.5 Differential Evolution Principles

DE works in cycles referred to as generations, very same as with the GE, as described earlier. The following steps takes place it order to evolve the best possible population of individuals:

1. **Definition of control parameters** – this is an obvious first step that involves setting of parameters described in the table 1 among other things such as the problem definition.

2. **Creation of initial population** – the evolution starts with the creation of an initial population so that the rest of the evolution has a set of individuals to work with. Vectors are generated in a completely random manner, same as in the case of GE.

3. **Cycle initialization** – in this phase, each individual of a current population is chosen and the following steps are applied to them:

4. **Evolution cycle** – in this step the mutation and the crossing takes place (in this order). Both processes have been described earlier. At the end, a new population of individuals is created.

5. **Evaluation of a terminating condition** – if the evolution satisfies its purpose (i.e. the given number of generations have been evolved), it stops. This is the only

terminating condition used by the original DE but other implementations may use additional rules. Typically, an evolution should stop once the perfect solution is found.

6. **Visualization** – if the evolution process is being visualized, the visualization should take place in this step. Then the evolution continues by step 3.

Exact details of evolution process are shown in [9]. Although parameters usually remain constant through whole evolution with the classical DE, it can also be implemented in a way that parameters do change depending on the evolution's own state. Such technique is referred to as meta-differential evolution and we can think of it as of differential evolution applied on differential evolution [8] [10].

4.6 Grammar Differential Evolution

Grammar evolution is traditionally related to genetic algorithms that act as the core that manipulates with genetic information. This thesis, however, deals with an idea of replacing the genetic part by a DE that would serve as a core for manipulating the genetic information [11]. However, since DE only works with vectors of (typically) real numbers, there is a need for an extra layer on top of it that would allow integration with the grammar part. This technique is referred to as Grammar Differential Evolution.

"Grammatical Differential Evolution (GDE) adopts a Differential Evolution learning algorithm coupled to a Grammatical Evolution (GE) genotype-phenotype mapping to generate programs in an arbitrary language. The standard GE mapping function is adopted with the real-values in the vectors being rounded up or down to the nearest integer value, for the mapping process. In the current implementation of GDE, fixed-length vectors are adopted within which it is possible for a variable number of elements to be required during the program construction genotype phenotype mapping process. A vector's values may be used more than once if the wrapping operator is used, and in the opposite case it is possible that not all elements will be used during the mapping process if a complete program comprised only of terminal symbols is generated before reaching the end of the vector. In this latter case, the extra element values are simply ignored and considered introns that may be switched on in subsequent iterations." [11]

5 SOMA

SOMA is an evolutionary technique invented by Prof. Zelinka in 1999. It is based on vector operations, same as with DE or PSO. The second is out the scope of the thesis. With SOMA, there is a difference in biological analogy. While most other evolutionary techniques evolves new generations while old ones dies out, SOMA uses still the same set of individuals that never extinct neither they are replaced by another set of individuals. Instead, SOMA represents a cooperative searching in a landscape (state space). Hence, it can also be qualified as a swarm intelligence algorithm. Hence iterations are referred to as migration cycles instead of generations in case of SOMA [8] [12].

5.1 Control Parameters

Same as other evolutionary algorithms, SOMA also uses a set of control parameters that affects the execution. Control parameters are listed in the table 2.

Parameter	**Recommended Range**	**Description**
PathLength	$\langle 1.1; 5 \rangle$	Path length determines how far an individual travels relatively to its distance from the dominant individual. If set to a value lower than 1, the individual would stop before the dominant individual, causing stagnation. Hence, it is recommended to use values larger than 1.
Step	$\langle 1.1; PathLength \rangle$	Determines the length of a single jump that an individual makes on its migration path. The step size is measured relatively to the path length. High step size values speeds-up the execution if the optimized problem is simple, preferably an unimodal function. If the function shape is more difficult or is not known at all, lower value are more appropriate. It is important to set the step in a way that it is not a multiple of PathLength. The evolution performance would degrade as individuals would stick to the dominant individual, preventing them to escape a local extreme.

PRT	$\langle 0; 1 \rangle$	Represents a perturbation rate. Perturbation affects direction of migrating individual in a way that it may block certain dimensions. The higher perturbation, the higher probability that individual's movement in given dimension will be blocked.
D	$\geq$ 1, depending on problem definition	Problem dimensions. Represents the number of unknown variables of the problem being optimized. Hence, it is only affected when the problem is changed or redefined.
PopSize	≥ 10	The population size. It has the same meaning as with other evolutionary algorithms.
MigrationCycles	≥ 10	Represents the number of migration cycles to be made before the evolution stops. It is an analogy of generation count as used by other evolutionary techniques. An optimal value is dependent on the problem definition.
MinDiv	Depending on a problem definition	RMinimal diversity is a terminating parameter. It represents the minimal diversity of fitness values between the best and the worst individual that once reached, the evolution stops. The optimal value is totally dependent on problem definition and no recommendation can be made in general. If set to negative value, the terminating condition is never satisfied. This may be desirable in some cases [8].

Table 2: Description of control parameters

5.2 Mutation

Same as with other evolutionary techniques, mutation is a fundamental part of SOMA. In case of SOMA, mutation is referred to as perturbation, since the perturbation vector is involved. Another reason is again biological analogy. It does not make sense to mutate individuals travelling over a landscape while it makes sense to affect their direction. The perturbation strength is determined by the PRT control parameter as described in table 2. It is roughly equivalent to the cross rate parameter used with DE. A perturbation vector is generated as follows:

1. For each problem dimension used, a random number is generated.

2. If the generated number is lower than the PRT parameter, 0 value is assigned; otherwise 1.

Once the perturbation vector is generated, it is used to affect the individual's migrating trajectory. It is multiplied by the vector of individual's planned movement. As a result, individual's movement in each dimension where perturbation vector contained 0 values is blocked [8].

5.3 Crossing

With other evolutionary techniques such as GE or DE, crossing represents creation of a new individual. SOMA is again different in this case but quite similar to PSO (which is, however, out of scope of the thesis). While an individual is migrating, the path is examined in several steps (depending on Step control parameter described in table 2). A series of individuals is generated and the best of them is preserved. Individuals' migrations adhere to the following expression:

$$\mathbf{r} = \mathbf{r_0} + \mathbf{m}tPRTVector, t \in \langle 0, PathLength \rangle \tag{3}$$

More precisely in [8]:

$$x_{i,j}^{ML+1} = x_{i,j,start}^{ML} + (x_{L,j}^{ML} - x_{i,j,start}^{ML})tPRTVector_j, t \in \langle 0, PathLength \rangle \tag{4}$$

5.4 SOMA Principles

As noted earlier, SOMA works in iterations called migration cycles which have the same role as generations used by, such as GE or DE. If a better position (i.e. solution with better fitness value) is found during the migration, the individual occupy that location and leave the previous. The process is shown on figure 7 where the left diagram represents the state before the migration and the right diagram after migration.

The following steps takes place it order to evolve the best possible population of individuals (or in this case, to migrate individuals into best locations possible):

1. **Definition of control parameters** – this is an obvious first step that involves setting of parameters described in the table 2 among other things such as the problem definition.

2. **Creation of an initial population** – the evolution starts with the creation of an initial population so that the rest of the evolution has a set of individuals to work with. Vectors are generated in a completely random manner, same as in the case of GE.

3. **Migration cycle** – Each individual is evaluated by the fitness function in order to determine which one is the leader. Selection of the leader in an important step because it determines the direction towards which the other individuals migrate. Individuals migrate towards the leader in steps defined by the control parameter. Their fitness value is re-evaluated after each "jump". If it is better that the original

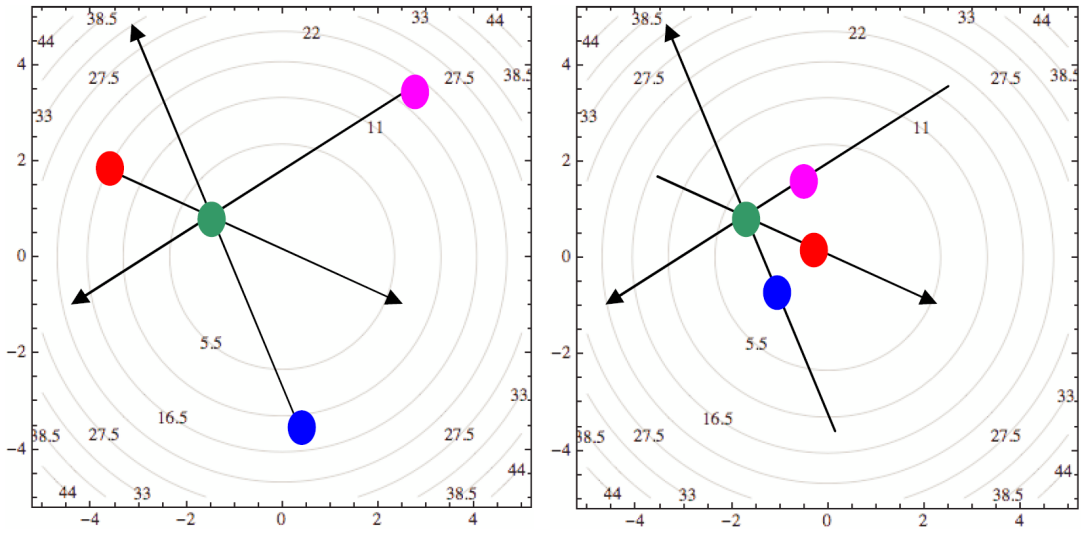


Figure 7: SOMA Migrations[8]

fitness value, the individual moves to the newly found position and leaves its old spot. The individuals travel up to the PathLength distance, which is another control parameter. At the end of migration cycle, all individuals except of the current leader had been moved or had at least tried to do so and had eventually remained in their original spots.

4. **Evaluation of terminating conditions** – in this phase a decision whether or not to continue by another migration cycle is made. The evolution is terminated when the difference between the best and the words falls below MinDiv control parameter. If the decision to make another migration cycle is made, the evolution continues by step 3.

5. **Termination** – The individual with the best fitness value is returned as a result [8].

5.5 SOMA Variants

There are currently five SOMA variants and two of them have been implemented in a program that is a part of the thesis and so are briefly discussed below. Description of the other variants can be found in [8].

- **All2One**. This is the basic strategy that has been described in the previous section. The basic characteristic of this variant is that all individuals migrate towards (and beyond) the leader.

- **All2All Adaptive**. There is no leader in this variant. All individuals migrate towards (and beyond) all other individuals. Unlike with All2All (which is not discussed here), the migrating individual does occupy a new position immediately after migrating towards each individual.

6 Randomness in Evolution

Evolution process consumes huge numbers of random numbers. Starting from the very beginning of the evolution – the generation of a random initial population, random numbers are consumed through whole evolution process. Selection, crossing and mutation are just few examples. Crossing strategies used with genetic algorithms, for example, may use random numbers to choose a certain point in two genotypes, after which the genetic information shall be swapped. Genetic selection strategies (like Roulette-Wheel Selection or Rank Selection) use them to choose individuals who will be given a chance to reproduce. Another example is mutation strategies which may use random numbers to determine which parts of genetic information are to be changed.

Evolutionary algorithms use rather PRNGs over RNGs for practical reasons. They are usually much faster in terms of performance and scales-out easily. Usage of PRNGs might raise some security concerns when used in cryptographic applications, e.g. However security is not an aspect that we would care about regarding evolutionary algorithms.

Since there is a part that deals with an idea of replacing traditional implementations of PRNGs such as Mersenne twister by a chaotic random number generator based on logistic map in the thesis, there is also a brief theoretical part devoted to chaos [8] [2].

7 Random Number Generators

A random number generator is a software algorithm or a physical device designed to generate a sequence of numbers that appear to be random and lack any obvious pattern. The needs of random numbers have led to the development of many methods for generating them. The oldest of them include dices, coin flipping or playing cards and dates back to ancient times. Nowadays these methods remain in use mainly in gambling as they obviously would not be very practical for modern computational applications.

There are two main classed of RNGs that serve to the same purpose but works on different principles:

- "True" random number generators.
- Pseudo-random number generators, which also includes chaotic generators [13].

7.1 "True" Random Number Generators

This class of generators is used mainly in application where high unpredictability of generated numbers is required. They generate numbers by measuring some physical phenomenon that is believed to be unpredictable by any means currently known to science. An example sources includes atmospheric noise, cosmic background radiation or radioactive decay. Unfortunately, measuring these phenomena requires extra hardware devices, commonly not present in ordinary computers. This limits the usage to applications where security is an extremely serious concern or where required by law. An example is on-line casinos. Many of them have a page at their websites that reveals more about RNGs they use, just in case of curiosity.

However, even an ordinary computer can access many sources of entropy that are random enough to be used by a software implementation of a RNG. An example source includes microphone noise, network traffic, battery level, fan speed, system temperature, accelerometer or gyroscope inputs if such hardware is present and many others. Even user input can be used as a source of entropy. It is possible to track mouse or keyboard events combined with the time of occurrence. Despite the wide range of possible sources that an ordinary computer can use, numbers generated this way are still considered a lower-quality compared to those generated by a specialized hardware device. Nevertheless, they are good enough for almost all real-world applications, including those concerning security (unless security concerns are extremely high).

An alternative way to obtain high-quality random numbers or just random raw data is via an on-line service such as [14], which provides various APIs for this purpose. Such data can be used in applications where security isn't a concern such as statistics or simulation. It is however unsuitable for cryptographic purposes due to the fact that the same data is available for everyone. Even if the data was generated exclusively for the API client, it still needs to be transferred over a public network which opens a possibility of interception.

The word "true" in the name of this chapter is quoted because the "true randomness" of RNGs is meant more in a sense of unpredictability than that the measured phenomena

would be really random in purest sense. Even radiation or atmospheric noise follows the physics laws. It just isn't possible to observe such phenomena in a level of detail sufficient for predicting its behavior. Furthermore, it is an open question whether or not does the true randomness even exist [15] [13].

7.2 Pseudo-Random Number Generators

Pseudo-random number generators serve the same purpose but operate in a different manner. In the initialization phase they need to be supplied a random data called seed or an initialization vector. That data is used to set PRNG's internal state. From that point on it generates numbers using a mathematical formula in a completely deterministic manner. Given the same seed or the IV, a PRNG generates still the same series of numbers. That has some implications:

- There is a need to obtain a random number to initialize the PRNG.
- The number of different series that a PRNG can generate is finite. Its maximal theoretical value is determined by the seed length. For instance, is the seed is a 32-bit number, then there is a theoretical limit of 2^{32} different series. In practice, it may be less if there are seed values that result in the same series being generated.
- Depending on the algorithm used, it is possible to predict the PRNG behavior after a sufficient number of previously generated values is collected.
- The generated series are periodical. The exact period differs and is an attribute of concrete PRNG implementation [13].

As previously noted a PRNG need to be initialized by a random data which is where an instance of the chicken or the egg dilemma arises. Where do we get a random seed before we have a random number generator? This problem is commonly addressed by two ways:

1. Either a different generator that does not require seeding is used to generate the seed for the original PRNG:

```
var initializingRng = new System.Security.Cryptography.RNGCryptoServiceProvider();
byte[] randomBytes = new byte[sizeof(int)];
initializingRng .GetBytes(randomBytes);
int seed = BitConverter.ToInt32(randomBytes, 0);
Random rng = new Random(seed);
```

Listing 1: Initialization of RNG using another RNG

2. Or the number of milliseconds that the system is running is used for seeding:

```
Random rng = new Random(Environment.TickCount);
```

Listing 2: Initialization of a RNG using a timestamp

The main advantage of PRNGs is performance which is usually much higher than in the case of RNGs. While RNG performance depends on the rate at which it is able to collect random entropy, PRNGs do not collect any more entropy once initialized. This allows their performance to scale-up with processor speed. Also, it can be easily parallelized. Due to independency on an external entropy input, each processor core can run a process with its own instance of a PRNG. Parallelization of RNGs would also require scaling-out their sources of entropy [13].

7.3 Mersenne Twister

The Mersenne twister is an example of a PRNG. I have chosen to describe this one because it is, by far, the most widely used PRNG [16]. It is also one of the generators available in my implementations of grammar evolution. The reason why Mersenne twister had become so popular may be because it addresses the issues of PRNGs used earlier such as RANDU [17]. It was developed in 1997 by Makoto Matsumoto and Takuji Nishimura. Nowadays the Mersenne twister is a default PRNG implementation in many programming languages.

There are more versions of Mersenne twister. The commonly-used one, MT 19937, which produces sequences of 32-bit numbers, has the following advantages:

- It has a very long period of 2^{19937}–1. That is where the variant name is derived from.
- The generated numbers are distributed with good uniformity.
- It passes many statistical randomness tests such as Diehard tests or NIST Test Suite [18].

Despite its numerous advantages and overall quality, it is not suitable for cryptographic purposes due to the fact that observing a sufficient number of subsequent outputs allows predicting all future iterations. The exact number differs by implementation details but it is relatively small in general. In case of MT19937 the number is just 624 [19].

Since the thesis also deals with chaotic generators, it is worth to note that chaotic generators also classifies as PRNGs. Their chaotic behavior is just apparent but in fact, chaotic systems are deterministic. After all, the field of science that covers such topics is called “deterministic chaos”.

7.4 Measuring Quality of Random Number Generators

There are several techniques to measure quality of random and pseudo-random generators. The NIST test suite is a well-known example of such technique [18]. The need for measuring RNGs and PRGNs quality arises from the fact that a low-quality generator producing output with obvious patterns or some other flaws could potentially compromise security if used for cryptographic purpose. However, even for applications such as statistics or simulation where security is not a concern, low-quality random numbers source may degrade the expected result in some way.

There are also other well-known test suites such as TestU01 [20] or Diehard tests that serve the same purpose but differs in the exact test designs.

8 Deterministic Chaos

Chaos theory is a mathematical field of study with many applications is various disciplines such as physics, economics, engineering or meteorology. It deals with behavior of dynamic systems highly sensitive to initial conditions – an effect known as the butterfly effect. A small difference in initial conditions may totally change outcomes of such systems. Even through chaotic systems are deterministic without any randomness involved, the level of sensitivity to their initial conditions makes them practically unpredictable.

Weather is a common example of a dynamic system with chaotic behavior, which makes its prediction challenging and not very reliable. Using the butterfly effect analogy, a butterfly might flap its wings, changing the initial conditions by producing an air which forms a stream by affecting other air currents and in the end, this tiny interference may change a direction of a tornado in the other end of the world.

Although definition differs in various publications, where are three common preconditions that a particular system must meet to be considered chaotic:

- It must be highly sensitive to initial conditions.
- It must be topologically mixing, meaning that as the system evolves in time, any two given regions of its phase state will eventually overlap.
- It must have a dense set of periodic orbits. Every point in the space is arbitrary close to some periodic orbit [15].

There is an application of the chaos theory it the thesis. As described in the following chapter, logistic map is used as a random number generator.

8.1 Logistic Map PRNG

Logistic map is one of the simplest dynamic systems that classifies as chaotic. It is often used as an example of how complex, chaotic behavior can arise from very simple models. The logistic map is defined as follows:

$$x_{n+1} = r x_n (1 - x_n) \tag{5}$$

It has been created in order to simulate situation in a natural environment occupied by two different animal species where the first species is hunted down and eaten by another species. Logistic maps capture the situation when the species that is being eaten dies out, freeing other resources in the environment, which allows the second species to occupy bigger portion of the environment. However, the extinction of the first specimen eventually causes starvation of the second. Consequently, the second specimen begins to die-out, leaving more space for the first specimen etc. Hence, in the equation,

- x_n is a ratio of existing population to the maximum population possible at year n (referred to as "carrying capacity". It is a real number between 0 and 1.
- x_0 hence represents the initial ration at year 0.

When logistic map is used as demographic model, it has pathological problem that some initial conditions may produce negative numbers (that does not make sense as population size). However, it is not a problem when used to build a PRNG [21]. What might raise some concerns regarding such application, however, is the distribution of generated numbers which are far from uniform distribution. The PRNG based on logistic map is a subject of experiment in later parts of the thesis and as it turned out, the non-uniform distribution is not a problem [15].

Cobweb Plot

A cobweb plot is a technique that can be used to investigate the orbit of dynamical systems. It is intended for visualizing one dimensional iterated function, which makes it suitable for the visualization of the behavior of logistic map. The plot is constructed using the following steps:

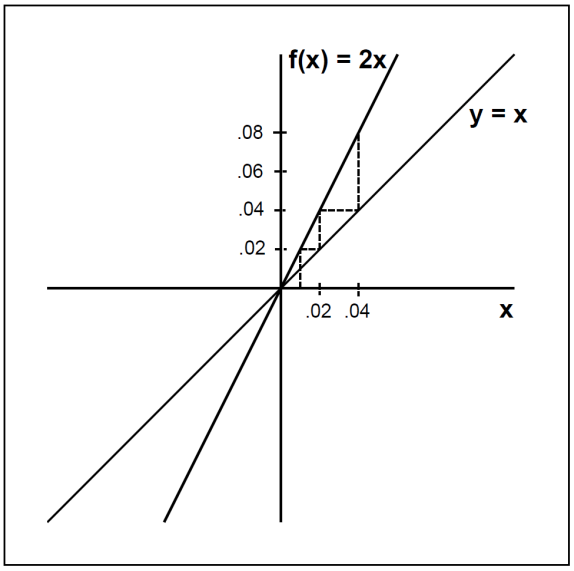


Figure 8: Cobweb plot

"For a map of the real line, a rough plot of an orbit—called a cobweb plot—can be made using the following graphical technique. Sketch the graph of the function f together with the diagonal line $y = x$. In Figure [8], the example $f(x) = 2x$ and the diagonal are sketched. The first thing that is clear from such a picture is the location of fixed points of f. At any intersection of $y = f(x)$ with the line $y = x$, the input value x and the output $f(x)$ are identical, so such an x is a fixed point. Figure [8] shows that the only fixed point of $f(x) = 2x$ is $x = 0$.

Sketching the orbit of a given initial condition is done as follows. Starting with the input value $x = .01$, the output $f(.01)$ is found by plotting the value of the function above .01. In Figure [8], the output value is .02. Next, to find $f(.02)$, it is necessary to consider .02 as the new input value. In order to turn an output value into an input value, draw a horizontal line from the input–output pair $(.01, .02)$ to the diagonal line $y = x$. In Figure [8], there is a vertical dotted line segment

starting at $x = .01$*, representing the function evaluation, and then a horizontal dotted segment which effectively turns the output into an input so that the process can be repeated. Then start over with the new value* $x = .02$*, and draw a new pair of vertical and horizontal dotted segments. We find* $f(f(.01)) = f(.02) = .04$ *on the graph of* f*, and move horizontally to move output to the input position. Continuing in this way, a graphical history of the orbit* $\{.01, .02, .04, \dots\}$ *is constructed by the path of dotted line segments."* [15]

On a finished plot, a rectangle would represent a period 2 behavior. The more complex closed loops the plot shows, the greater period. A truly chaotic function would eventually fill-out the whole area, indicating an infinite number of non-repeating values. The reason, why none of the diagrams 9-17 contain any area that appears filled-out by a solid color is because the function is iterated only finite number of times when plotting. Moreover, a number representation with finite precision is used in computational applications which make any dynamic system periodical. The presented cobweb plots reveal that the behavior becomes more chaotic as the parameter increases. Especially the value of 4 seems promising. However, the transition isn't linear. E.g. there is a pathological problem in certain intervals as shown in the following chapter.

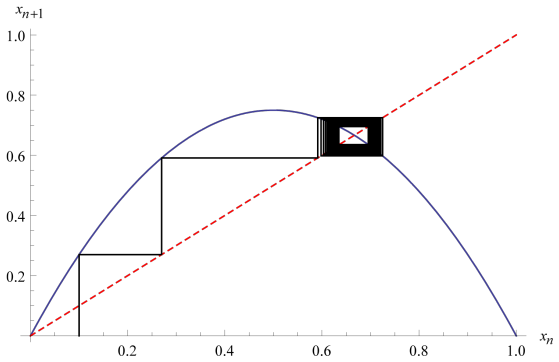


Figure 9: Cobweb plot for $r = 3.000$

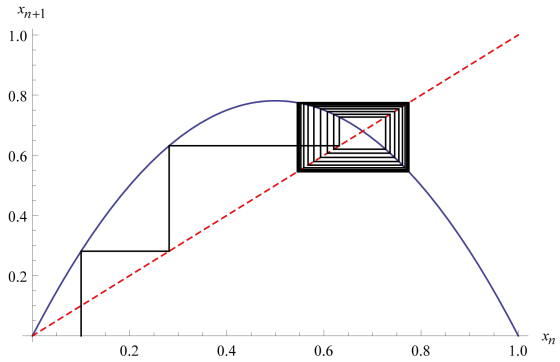


Figure 10: Cobweb plot for $r = 3.125$

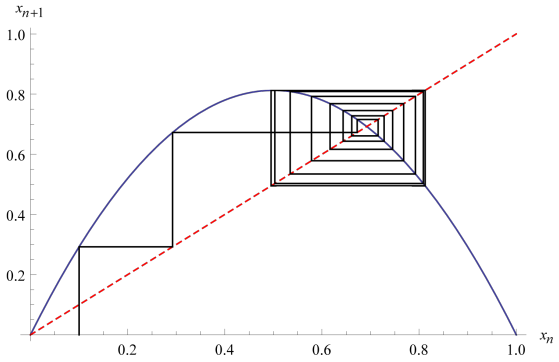


Figure 11: Cobweb plot for $r = 3.250$

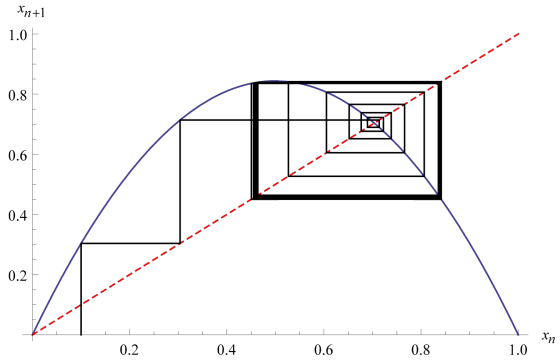


Figure 12: Cobweb plot for $r = 3.375$

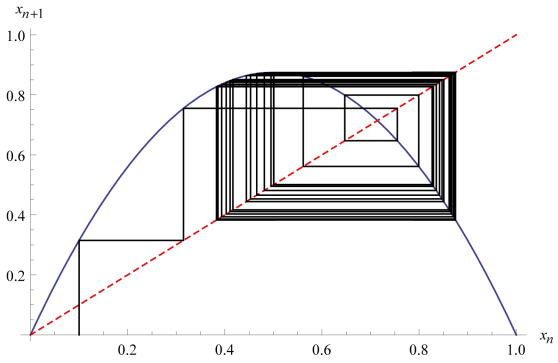


Figure 13: Cobweb plot for $r = 3.500$

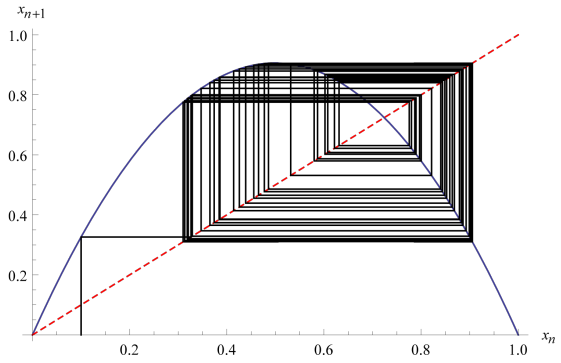


Figure 14: Cobweb plot for $r = 3.625$

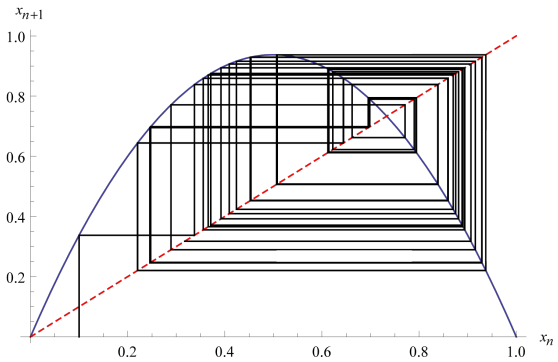


Figure 15: Cobweb plot for $r = 3.750$

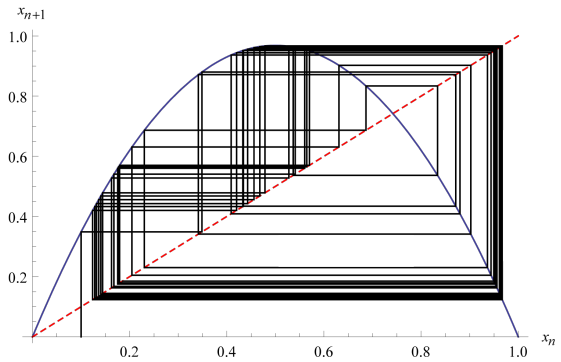


Figure 16: Cobweb plot for $r = 3.875$

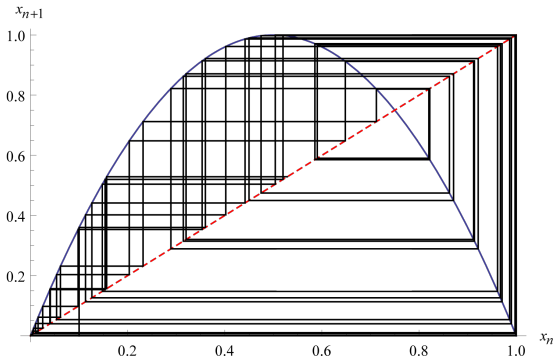


Figure 17: Cobweb plot for $r = 4.000$

Bifurcation Diagram

Bifurcation diagram is another tool for inspection of the logistic map's chaotic behavior. Unlike cobweb plot, it shows its behavior for a whole range of parameter (called bifurcation parameter) values at once. There is the bifurcation parameter on the horizontal axis and

the population values of the logistic function on the vertical axis.

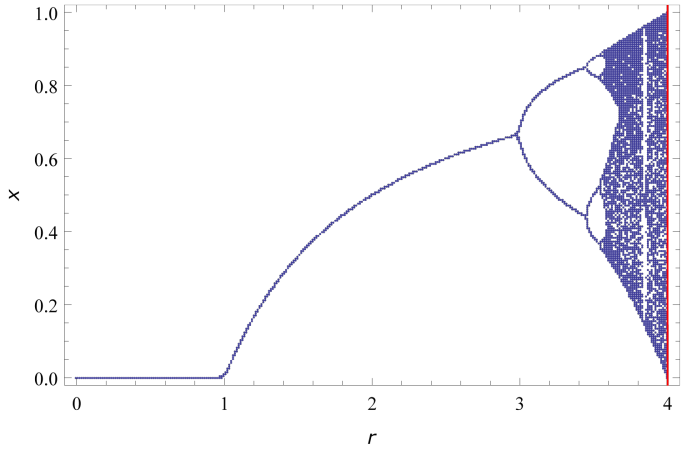


Figure 18: Bifurcation Diagram

Bifurcation diagram is made by a series of diagrams such as 18 showing which value the logistic map returns for given bifurcation parameter r. I.e. there is a plot for first 50 iterations for r = 4 on figure 19.

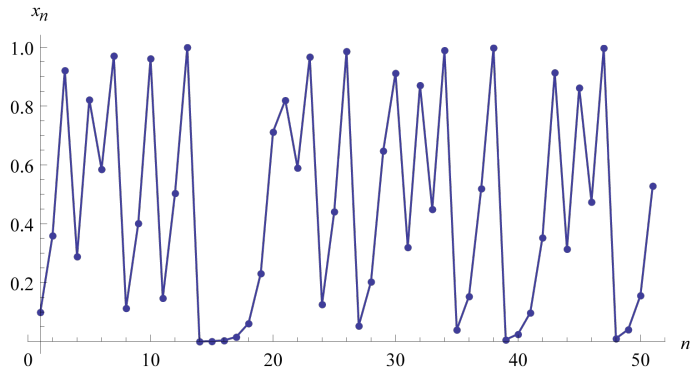


Figure 19: First 50 iterations for $r = 4$

A whole series of plots such as 19 is generated for r from 1 to 4, using some small sampling step. Each diagram of the series is then flattened into one-dimensional space and put next to another sample, which is put next to another sample etc. All flattened plots put side by side together compose bifurcation diagram 20. Hence, bifurcation diagram allows us to see ranges of possible values that are generated depending on the bifurcation parameter r. Since it is quite difficult to construct a bifurcation diagram using an application that does not support it natively, such as Microsoft Excel, I would recommend using some specialized mathematical software such as Wolfram Alpha or Mathematica.

Looking at the bifurcation diagram, we can see period-doubling which is a common behavior observable in chaotic systems [15]. The period of once stable system doubles until it eventually becomes chaotic. In the case of logistic map, we can observe a stable (thus not very useful) behavior for r up to approximately 3.8. Beyond this point, the behavior appears more or less chaotic. In later parts of the thesis an experiment with logistic map PRNG using $r = 4$ is presented.

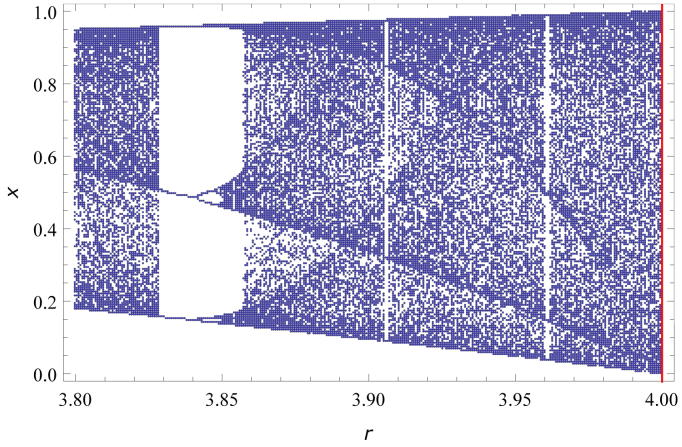


Figure 20: Zoomed bifuraction diagram

9 Grammar Evolution Implementations

This chapter describes implementation details of programs that I have made for the purpose of the thesis. I have made three different applications that serve the same purpose but each of them does so by using a different evolutionary algorithm:

- **ge.exe** that utilizes grammar evolution
- **gde.exe** using grammar differential evolution
- **gsoma.exe** obviously using grammar SOMA

9.1 Supported Platforms

Although the list of used tools and technologies may raise concerns about cross-platform compatibility, the implementation has been done with portability in mind. The code runs smoothly on both 32-bit and 64-bit versions of three major operating systems – Windows, OS X and Linux.

It is possible to run exactly the same compiled code on all three platforms. There is no need of compiling an individual version of each platform. This is enabled thanks to the fact that the code compiles into MSIL (an intermediate language comparable to Java bytecode) instead of native machine code. However, there is an obvious implication that it requires a runtime environment and will not run by itself. Since .NET Framework is only available on Windows platform, Mono has to be used on OS X and Linux. It is an open source project that aims to provide a runtime environment compatible with the original .NET Framework.

Despite Mono qualities, it is not 100% compatible. Some parts of the original framework are not implemented at all [22] (e.g. because of overwhelming complexity or because they are too platform-specific). Even features that have been implemented may behave slightly different in edge cases. Even if Mono was 100% compatible, application still have to take into account that operating systems differs in many things that Mono just cannot abstract. However, no modifications in the code have been needed to make in portable in my case. I believe this was because the code is very self-encapsulated. Its only interaction with the operating system involves the CLI and file system access which are both very common features. Nevertheless, I had to have the compatibility in mind when choosing external libraries.

9.2 Exploring the Source Codes

Opening the solution requires Microsoft Visual Studio 2012. I have been using the Ultimate edition obtained from Microsoft's DreamSpark Plus program. However, I believe that even the express edition which is publically available for free should be enough to experiment with the solution. Although I have been using some features that are not available in Express version, neither of them is needed to build the solution. That had been mostly instrumentation tool such as Performance Wizard or visual aids such as class

modelling. It should also be fairly easy to convert the solution for Visual Studio 2013 which is the latest version available at the time of writing of the thesis. The solution consists of 14 projects. Each serves the purpose described in the table 3.

Project	**Project Type**	**Purpose**
Thesis.Cfg	Class Library	Contains classes regarding context-free grammars and also a BNF parser.
Thesis.Cfg.Tests	Unit Test Project	Contains unit tests for Thesis.Cfg project.
Thesis.Demo.Approximation	Windows Forms Application	Contains an application with graphical GUI that I have created during my early experiments with GE.
Thesis.Demo.Approximation.Tests	Unit Test Project	Contains unit tests for the Thesis.Demo.Approximation project.
Thesis.Demo.SimpleSum	Console Application	This is another of my early experiments. Contains a program that tried to evolve a mathematical expression whose result equals 100.
Thesis.Gde	Class Library	A re-usable library containing GDE logic.
Thesis.Gde.Cli	Console Application	Contains a console application for interacting with the GDE library.
Thesis.GE	Class Library	A re-usable library containing GE logic.
Thesis.GE.Cli	Console Application	Contains a console application for interacting with the GE library.
Thesis.GE.Tests	Unit Test Library	Contains unit tests for the Thesis.GE project.
Thesis.GSoma	Class Library	A re-usable library containing GSOMA logic.
Thesis.GSoma.Cli	Console Application	Contains a console application for interacting with the GSOMA library.
Thesis.Shared	Class Library	Contains component shared among other projects. CLI utilities, fitness and mathematical functions, random number generator implementations as well as other things.

Table 3: Solution projects

9.3 Architectural Overview

My goal was to design a proper architecture of the solution which respects various architectural patterns and principles. There was a special emphasis on separation on concerns principle through the solution. The architecture aims to be clear and to support extensibility which also means that it is not strictly enforced in places where it would make sense for the current feature set but might prevent future extensions. There are some places when the architectural patterns are rather relaxed but it is always on purpose and never on the account of reduced code readability. The following chapters describes various implementation parts and highlight possible extension points. GE, GDE and GSOMA share large parts of common codebase.

BNF

The logic regarding CFGs is implemented in the Thesis.Cfg class library. The most complex part of this library is the parser found in BackusNaurForm.cs that reads a file in the BNF format and parses it into an in-memory representation of a CFG. Since the BNF format is quite simple, I have decided to create my own parser implementation.

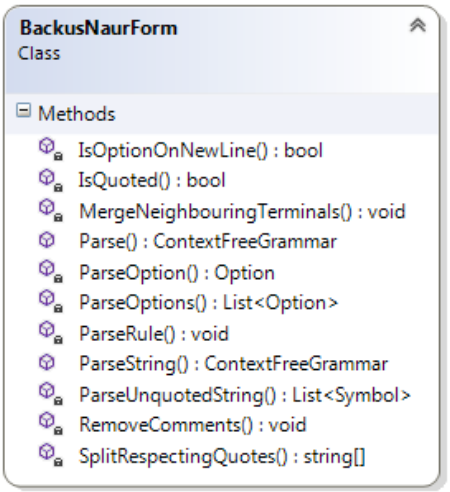


Figure 21: Backus Naur Form class diagram

It is not based on any other library such as ANTLR although it would be probably suitable for this purpose. In case of interest in the exact parsing workflow, the code is heavily commented and easy to read, in my opinion. Since the whole solution is multi-platform, the parser supports all three types of commonly used end-of-line characters (CRLF on Windows platform, CR on Linux and LF on OS X). Furthermore, the parser supports few extra features, not present in the classical BNF specification:

- **Comments.** Each must start on its own line and must start with one of the following characters: "#", "//" or ";". Multiple comment styles can be used in a single file.

- **Quoted strings.** It is possible to express characters such as "::=", "¡" that would normally have a special meaning (or no meaning in case of whitespaces) by encapsulating them into double quotes.

Personally, I consider my BNF parser implementation quite robust. The parses successfully passes a set of unit tests in the Thesis.Cfg.Tests namespace designed to test both common and edge cases. Please consider the following testing grammar, found in Thesis.Cfg.Tests.Grammars.ComplexGrammar.bnf:

```
# Ugly but syntactically correct grammar
# <S> ::= abc

<S_-a0>::="a|b'c" def ghi | <A> 'jk"l' <B> m n o
         |p
 |q
          // A comment
  ;
  #
     |r

  <A>::=<B>
<B>                ::=<EOL>
```

There are some tricky parts like mixing quote styles (double quotes inside of single-quoted strings and vice versa), quoted characters of special meaning, comments in the middle of rule definition, weird non-terminal name etc. Even such file is parsed correctly. The related unit test is performed by the ParseComplexGrammar method in Thesis.Cfg.Tests.BackusNaurTest.cs.

Context-Free Grammar

Another important part of the Thesis.Cfg library is a DerivationTree class which represents what the name says. It is used by all three evolutionary algorithms to build phenotypes out of genotypes. Although derivation tree is a hierarchical data structure, it is in fact represented by a linked list. It used to be implemented as a classical tree structure originally but I have been encountering strange program behavior when the recursion level reached certain point, usually around 100. Such situation may arise when a large number of rewrite rules is needed to successfully build an expression, which is quite common situation, given that some genotypes may even result in infinite derivation trees. The program would not crash or throw an exception is such situation. It would just stall which made it hard to localize where the problem was. Current implementation using linked list do not suffer from deep recursions and in addition, offers many times better performance which is also considerable improvement, given how often a phenotype is being built during evolution execution.

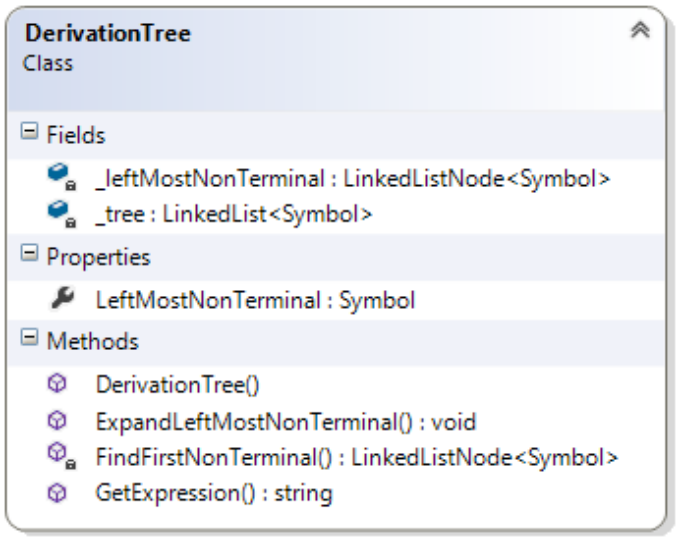


Figure 22: Derivation Tree class diagram

Fitness Functions

The logic related to fitness evaluations is found in the Thesis.Shared.Fitness namespace. There are two fitness functions already implemented and others can be easily added, thanks to the modular architecture. In order to implement new fitness function it is enough to make it implement the IFitnessFunction interface. There have been two fitness functions implemented, both designed for the fitness fitting problem. Both use the NCalc library for evaluation of mathematical expressions.

- **DevSumFitness** computes the sum of absolute deviations for all discretized function samples.
- **RSquaredFitness** works on the same principle but computes R^2 deviation instead of absolute deviation.

Both functions were successfully unit-tested and are fully functional.

Fitness Caching

I have noticed that it is common for all evolutionary techniques implemented, disregarding control parameters, that the same phenotypes gets evaluated by a fitness function all over. That led me to an idea of implementing a cache that would remember certain number of recent phenotypes and corresponding fitness values. If the case was hit, it would return the fitness value instantly without evaluating the actual logic. The cache is implemented as I hash map where phenotype is the key and its fitness evaluation is the value.

Algorithmic complexity of various fitness functions may vary considerably, not mentioning that it is even possible to construct a fitness function that depends on input from

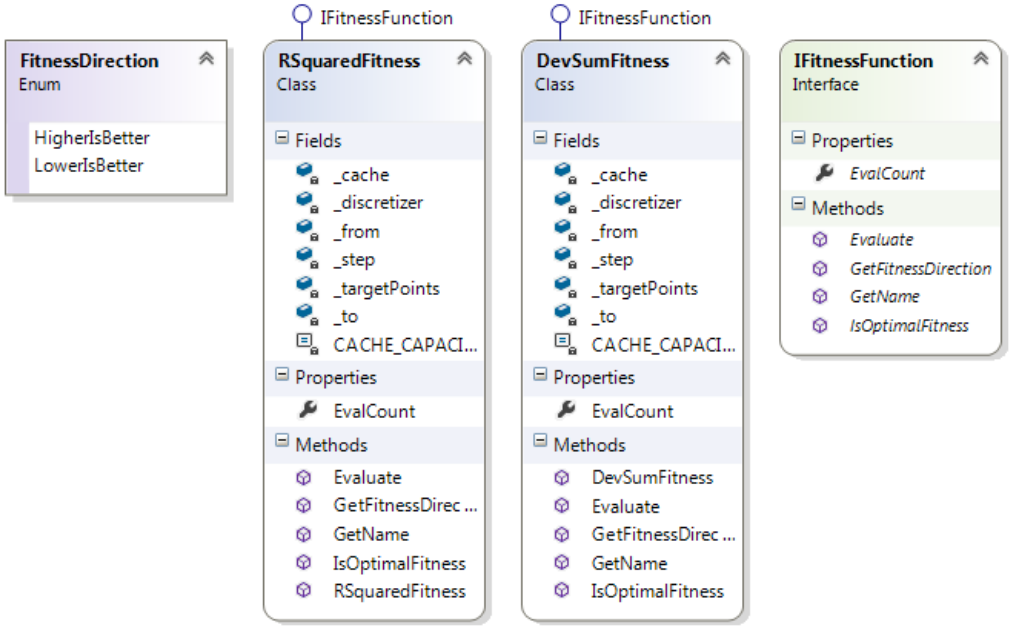


Figure 23: Fitness evaluations class diagram

an external system or from a human operator, which may be time-demanding. However, even if the fitness function was purely software component with simple logic, looking-up a value in a hash map should be much faster anyway, even for very large hash maps. Although I had not have done any experiment measuring the impact of caching on the overall evolution performance, it is evident that it speeds-up the execution dramatically. For demonstration, it is easy to comment-out the line of code that adds a record to the cache so it can never be retrieved. A huge performance slow-down can then be observed.

Using the cache, it is common that the evolution performance measured by generations or migration cycles per second increases exponentially as the evolution converges to certain solution. The higher convergence, the lower numbers of individuals have mutually different phenotypes. Thanks to it, still higher percent of the phenotypes is cached, avoiding execution of the actual fitness logic which consequently causes rising of performance. If we notice such dramatic performance increase, it is usually a sign that the divergence of individuals in population is very low and that the evolution has converged to certain solution.

Please note that it is completely wrong to compare performance of any two evolutions by counting generations or migrations cycles evolved over a period of time, as explained in earlier parts of the thesis! It only makes sense in this particular context, regarding caching functionality.

9.4 GE Implementation

The logic regarding GE implementation is placed in three projects:

- Thesis.GE which is the main GE core.
- Thesis.GE.Cli, the console application.
- Thesis.GE.Tests containing unit tests for classes from the Thesis.GE namespace.

Thesis.GE is probably the most interesting of them as is contains the actual GE logic, which is further described in following chapters.

Selection

The selection logic is placed in the Thesis.GE.Selection namespace. There have been three selection strategies implemented (see fig. 24). Since GE use selection strategies through the ISelectionStrategy interface, it would be easy to extend the application by another selection strategy if needed. Selection strategies logic adheres to principles described in the theoretical part of the thesis. The code is quite straightforward and heavily commented so I believe that no further explanation is needed.

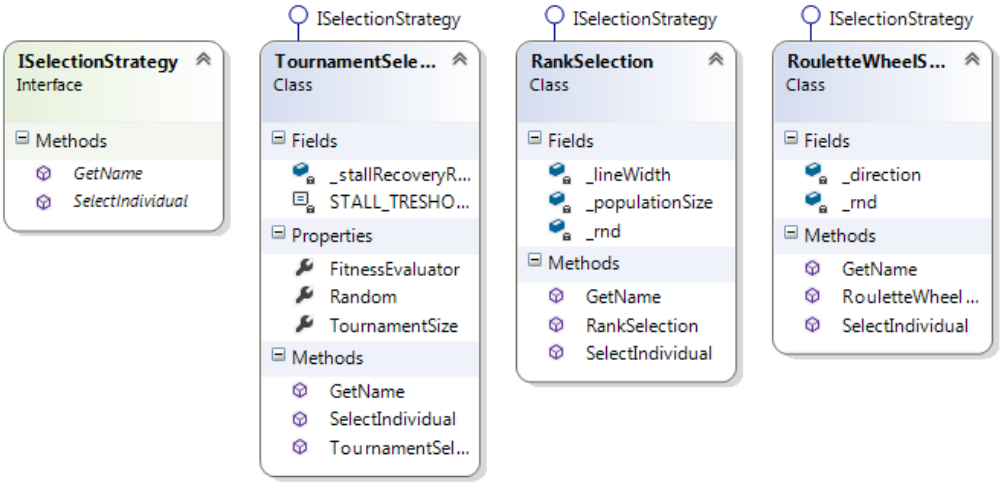


Figure 24: Selection class diagram

Crossing

The crossing logic is placed in the Thesis.GE.Crossing namespace. Same as in the previous chapter, the rest of application use crossing through the ICrossingStrategy interface (see fig. 25) which makes implementation of new crossing strategies easy. The strategy implemented in the RandoPointCrossingStrategy class represents a single-point strategy and works on principles describer in the theoretical part of the thesis.

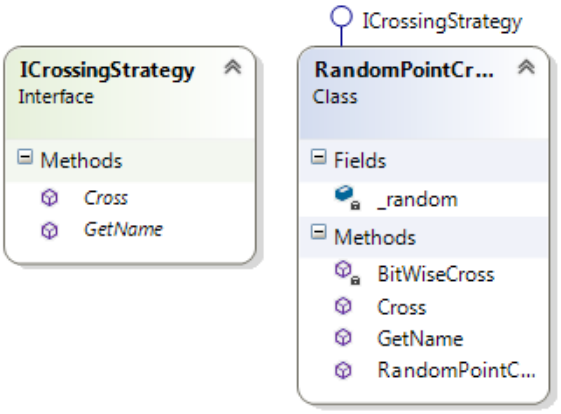


Figure 25: Crossing class diagram

Mutation

The mutation logic is placed in the Thesis.GE.Mutation namespace and is also used through the IMutationStrategy interface (fig. 26) by the rest of the application. Given the mutation strength parameter, the RandomMutationStrategy uses Poisson distribution to determine the number of bits to swap. The use of Poisson distribution is actually important in this case. If the number of bits to swap was determined as chromosome length multiplied by mutation strength, the result is zero in most cases, given that both parameters use common values. Increasing such values to one is not ideal solution either because then the mutation would be stronger that desired in average.

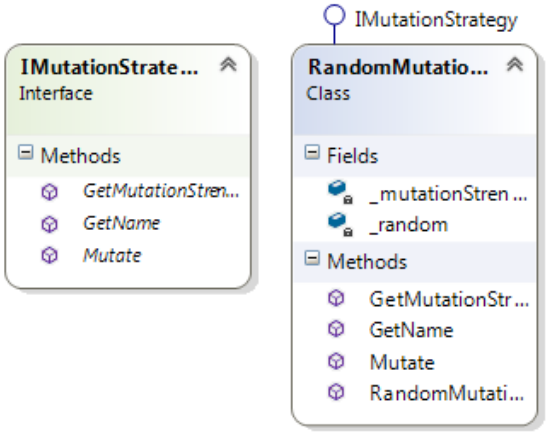


Figure 26: Mutation class diagram

9.5 GDE Implementation

The logic regarding GDE implementation is placed in two projects:

- Thesis.GDE which is the main GDE core.
- Thesis.GDE.Cli, the console application.

Since GDE is far less modular that GE, it has been implemented by a single class. Despite this, the code is quite short and readable. DE is very simple algorithm and so is GDE that enhances it by the grammar layer. There is actually just one aspect in the code worth highlighting. In the Iterate method, there is logic that constraints trial vector values in order to prevent them from escaping the state space. For each dimension that reach out of valid range a new value from within the valid interval is generated randomly. Another approach would be to shift the value to the edge of the interval.

9.6 GSOMA Implementation

The logic regarding GE implementation is placed in two projects:

- Thesis.GSoma which is the main GE core.
- Thesis.GSoma.Cli, the console application.

There have been two GSOMA variants implemented:

1. All-to-All Adaptive, represented by the GSomaAllToAllAdaptive class
2. All-to-One, represented by the GSomaAllToOne class

As shown on fig. 27, both variants inherits the common GSomaBase base class that contains parts that are common for both variant which is actually almost everything except of the MigrateIndividual abstract method, that have to be overridden by the inheriting class. That is because each GSOMA variant uses different migration strategy, as described in the theoretical part of the thesis.

9.7 Logistic Map PRNG Implementation

The Thesis.Shared.Random namespace contains the LogisticRng class which is the implementation of chaotic PRNG based on logistic map. The second PRNG that is used through the thesis is not implemented anywhere in the solution as it is an integral part of the .NET Framework. Since the logistic map PRNG implementation inherits the System.Random class, it is easily interchangeable.

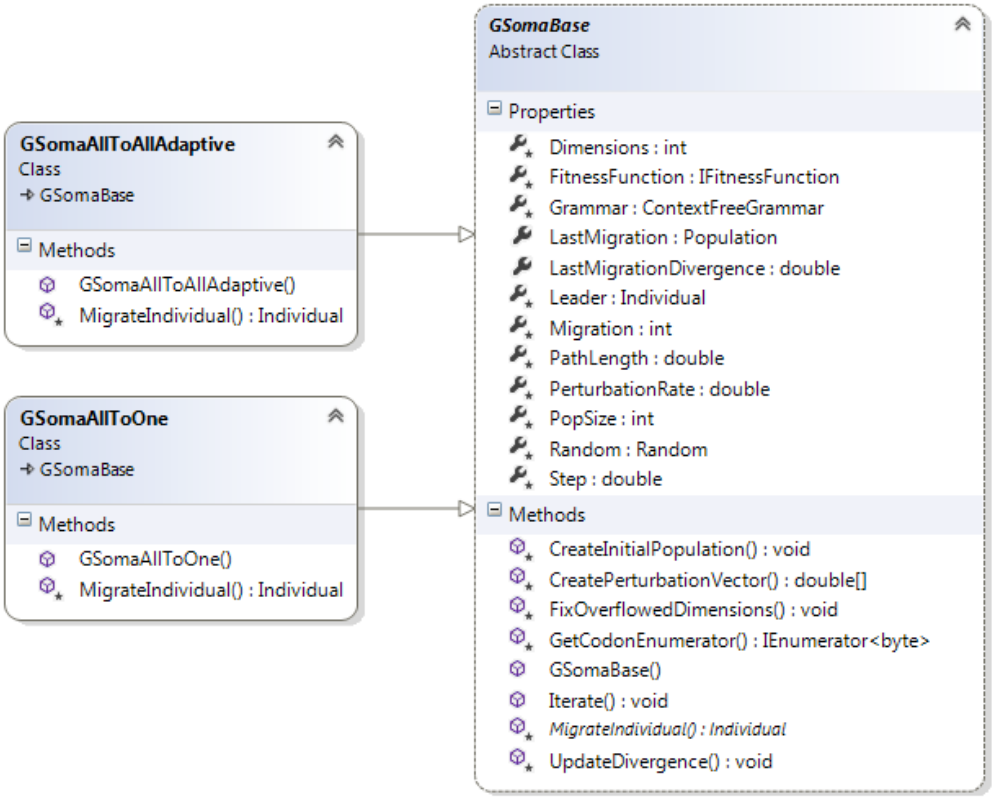


Figure 27: GSOMA class diagram

10 Comparison of GE, GDE and GSOMA Performance

Both DE and SOMA are well regarded for their performance when applied on various optimization problems not involving CFGs. Genetic algorithms, on the other hand, are considered less performing. That said there is an assumption that also the grammar-enabled variants of both (GDE and GSOMA) perform better than the classical GE. This experiment aims to measure the GE, GDE and SOMA performance and compare it mutually.

10.1 Experiment Design

Following paragraphs describes the problem definition and control parameters that were used in the experiment. Each test was repeated 50 times for higher accuracy.

Problem Definition

All three evolutionary techniques mentioned were used to optimize a function fitting problem. The goal was to find a function that describes given function as closely as possible. The function was discretized into 50 points on a given interval. Candidate solutions were evaluated as follows:

1. They were discretized the same way as the original function.
2. An absolute deviation was computed in each discretized point.
3. The cost value was determined as a sum of these absolute deviations.

An ideal solution would have a zero cost value while higher numbers means that the fitting was not perfect. That said no conversion was needed to transform cost values into fitness values as lower values already represented a better solution. Furthermore, the ideal solution is represented by zero which is the optimal case.

There were two functions used in this experiment to achieve more accurate results:

1. Sextic function: $x^6 - 2 \cdot x^4 + x^2$ (fig. 28)
2. Quintic function: $x^5 - 2 \cdot x^3 + x$ (fig. 29)

Both functions (fig. 28 and fig. 29) used parameters described in table 4.

Fitting Interval	$\langle -1; 1 \rangle$
Discretization Step	0.04
Discretization Points	50. Determined as l/d where l is the interval length (2 in this case) and d is the discretization step.

Table 4: Experiment design parameters

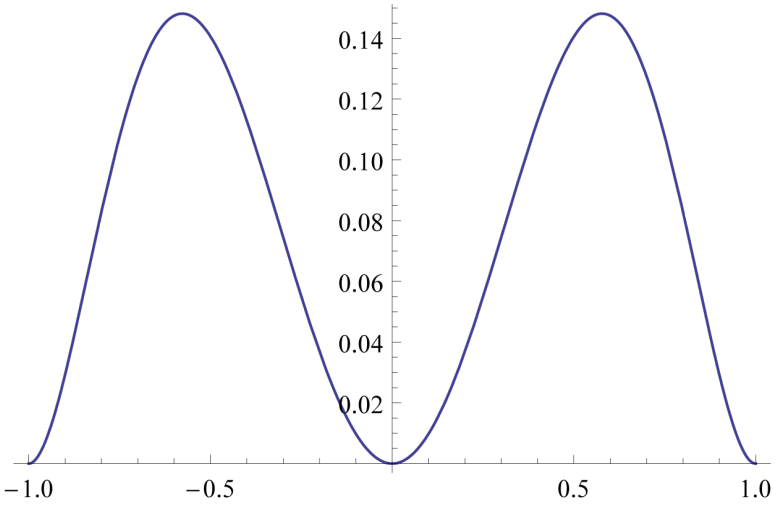


Figure 28: Sextic function

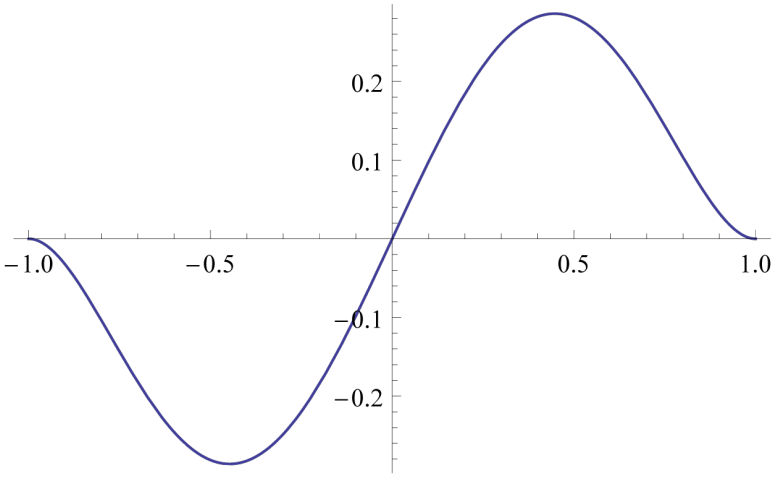


Figure 29: Quintic function

Control Parameters Settings

Different control parameters were used for each of the evolutionary techniques (GE, GDE and GSOMA). One reason is obvious – each of these techniques use a different set of control parameters. However, even if they would not, GE e.g. is known to require much bigger population sizes than DE to perform equally well so setting the same parameter values would not be fair even if it was technically possible. [ref Aplikace umělé intelligence] This is apparent with different GSOMA variants where highly different numbers of migration cycles are used, yet both variants evaluate the fitness function the same times. To deal with this problem, I had to find three different set of parameters that gives the best results possible for each individual evolutionary technique. These parameter sets

(described in tables 5, 6, 7, 8) were used for the experiment.

Parameter	**Value**
Cross Rate	0.85
Selection Strategy	Roulette-wheel selection
Population Size	300
Chromosome Length	50
Generation Count	500
Elitism Level	1

Table 5: GE Control Parameters

Parameter	**Value**
Cross Rate	0.85
Mutation Constant	0.7
Population Size	50
Chromosome Length	50
Generation Count	20000

Table 6: GDE Control Parameters

Parameter	**Value**
Path Length	3.5
Step Size	0.11
Perturbation	0.3
Population Size	50
Dimension (Vector Length)	50
Migration Cycles	1000

Table 7: GSOMA All2One Control Parameters

Parameter	**Value**
Path Length	3.5
Step Size	0.11
Perturbation	0.3
Population Size	7
Dimension (Vector Length)	50
Migration Cycles	20

Table 8: GSOMA All2All Adaptive

10.2 Grammar

The following grammar described by BNF has been used to build phenotypes in the experiment:

```
<expr>   ::= <expr><op><expr> | <number> | [x]
<op>     ::= + | - | * | /
<number> ::= <digit>.<digit><digit><digit><digit>
           | -<digit>.<digit><digit><digit><digit>
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The grammar is very general. It is not biased in any way that would help to help find a solution faster. It had been designed that way to make the problem more difficult to optimize.

Experiment Methodology

All three evolutionary techniques (including both SOMA variants) using settings as described earlier were used to optimize two different function fitting problems (also describer earlier). Output data were then transformed into two different forms:

1. The one, suitable for plotting burn down diagrams as shown it the results chapter.
2. Another one, allowing making column charts comparing their performance mutually.

All evolutions have been run in 50 iterations to increase the statistical confidence of presented results. Evolutions were evaluated based on the first 20,000 fitness function evaluations. Any evaluation beyond this limit did not affect the result in any way.

Experimental Environment

A Lenovo ThinkPad T420 laptop with the configuration as described in table 9 had been used to run the experiment. However, this information is present mainly because it is a custom to mention it. Since results are not affected by the execution time in any way, whatever machine capable of running involved programs should give similar results, sooner or later.

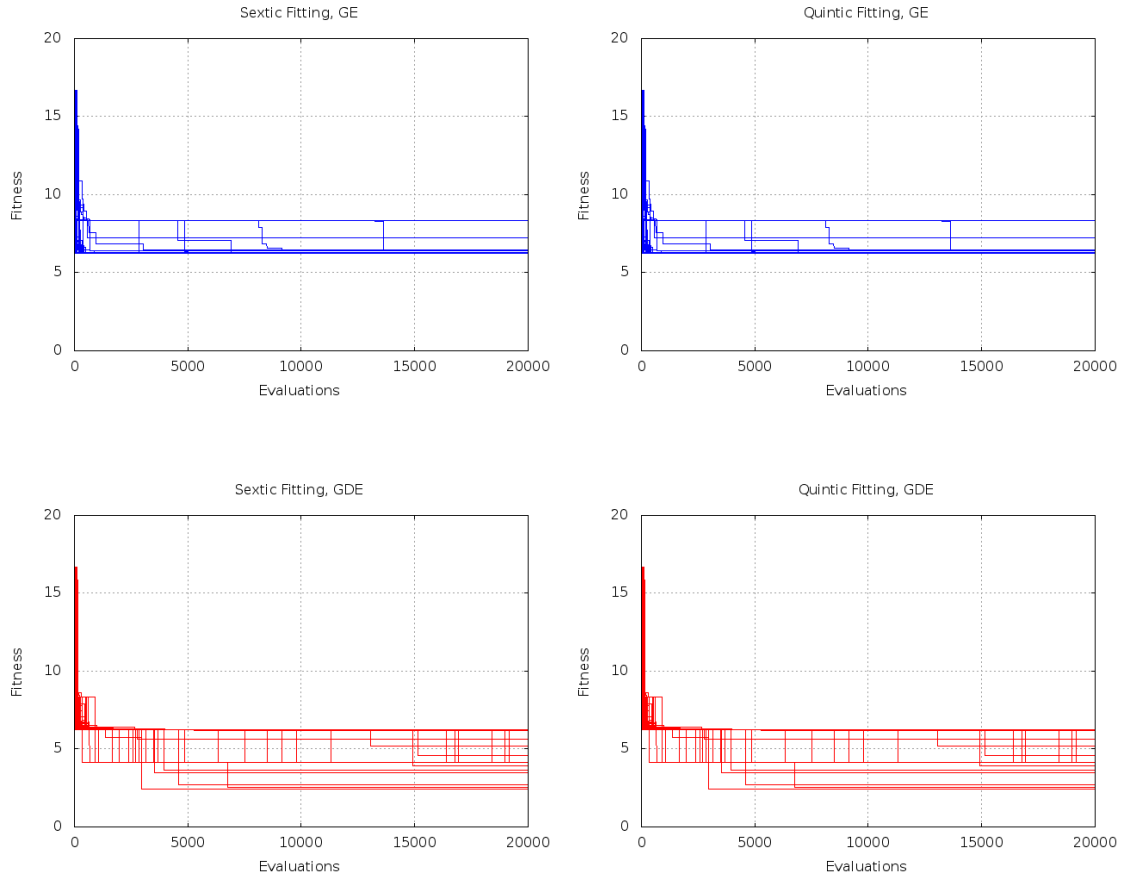
Operating System	Windows 7 64-bit SP 1
CPU	Intel Core i5-2430M @ 2.4 GHz • 2 cores • 2 threads per core
Memory	8 GB (2 $\times$ 4 GB @ 1333 MHz)

Table 9: System parameters

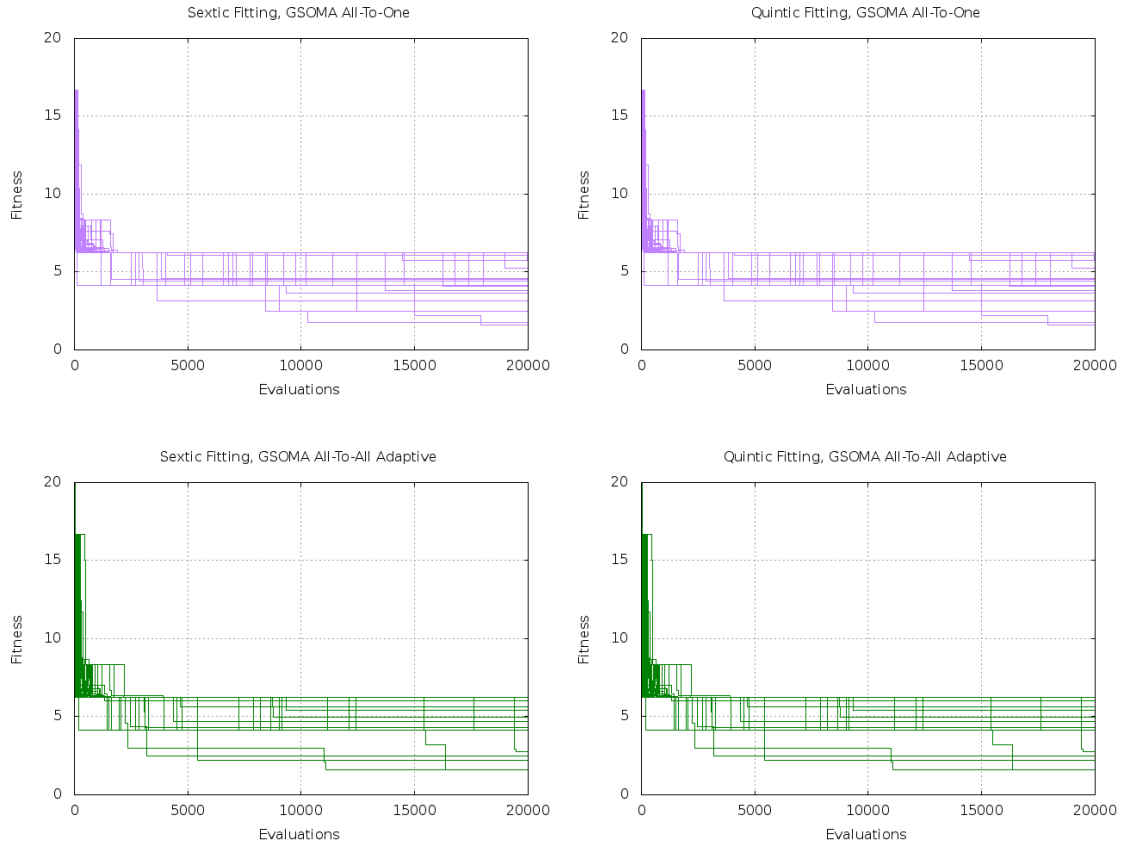
10.3 Experiment Results

This chapter contains graphs that visualize the experiment results in different ways. The first type of diagrams represents the evolution progress as it tries to approach the null fitness value. Many lines (specifically 50 but many of them overlap) can be seen in each of these diagrams. That is because each evolution have been run in 50 iterations as mentioned earlier and each chart visualizes these iterations all at once.

There are two diagrams for each evolutionary technique tested (GE, GDE and GSOMA). That is obviously because two different problems were optimized. The diagrams are quite self-describing. There is a fitness value on the vertical axis and a number of fitness function evaluations on the horizontal axis. The individual lines consequently represents how many times the fitness function had to be evaluated to find a solution having the particular fitness value.



The second type of diagrams shows the overall performance of each evolutionary technique, each of them being plot on the horizontal axis. The vertical axis shows the average fitness values of the best solution of each iteration that have been created using 20,000 fitness function evaluation at maximum.



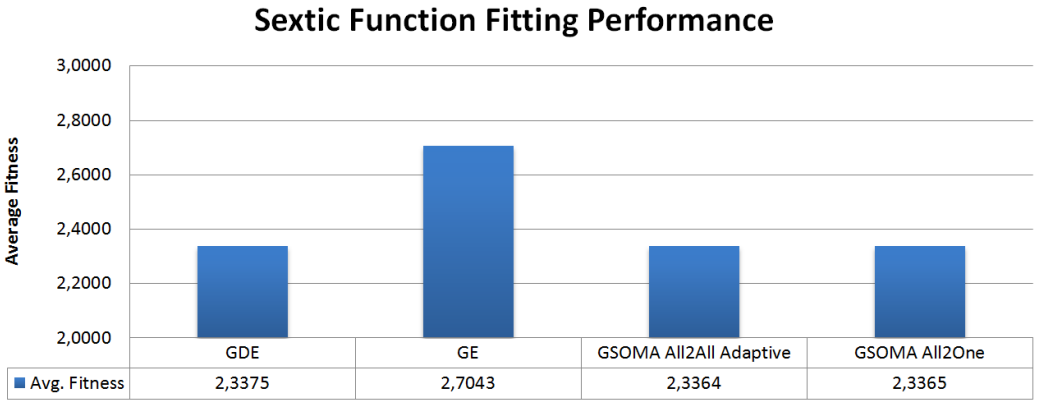


Figure 30: Sextic function fitting performance

10.4 Experiment Conclusion

Based on the presented results, we can accept the assumption formed earlier. Both GDE and GSOMA perform better that the classical GE based on genetic algorithms.

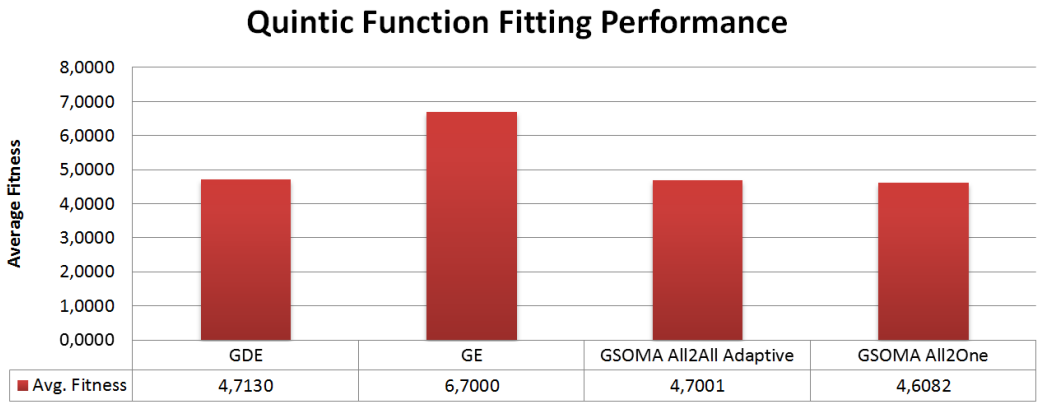


Figure 31: Quintic function fitting performance

Personally, I also consider both of them easier to implement than GE – especially the GDE is really simple. Given these two arguments, it makes no sense to use GE in its classical form which is both more difficult to implement and less performing. However, I believe that this experiment is not enough to doom the classic grammar evolution in general. Although this experiment's results are quite convincing, there may be some cases (maybe with problems other than function fitting) when GE might perform better than GDE or GSOMA. Furthermore, the experiment's results are highly dependent on control parameter values and as noted earlier, there is no way to tell what parameters should be set to make the comparison totally fair.

11 Logistic Map PRNG Suitability

The second experiment deals with the idea of replacing the default system PRNG (System.Random) by a chaotic generator based on logistic map. Although values distribution generated by the logistic map PRNG are far from uniform (as they tend to stick to either boundaries of given interval), there is an assumption that such generator should be feasible for use with evolutionary algorithms. This experiment aims to measure performance of the logistic map PRNG and to compare it with the default system generator. Following chapters describe experiment design and concludes results.

11.1 Experiment Design

The experiment re-uses some parts of the first experiment's design. In particular, the following design parts remain the same:

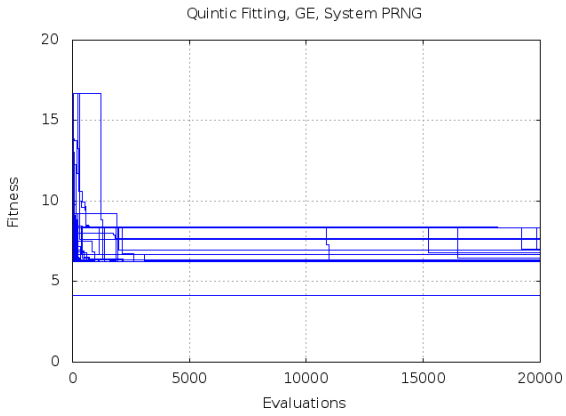
- The problem definition
- Control parameters
- The grammar
- The experimental environment

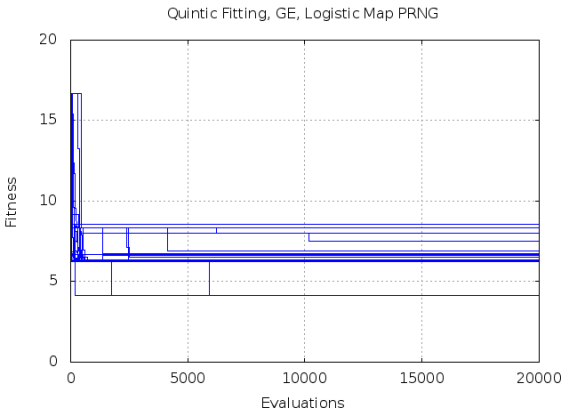
For details, please refer to according chapters of the first experiment. There is a small difference in the methodology. Unlike with the first experiment, evolutions have been run with both systems PRNG and the PRNG based on logistic map. That doubles the number of graphs produced. The aim in this case is to compare performance using the two PRNGs rather than compare different evolutionary techniques to which other. GE using the default system PRNG is compared only to the GE with logistic map. The same applies for GDE and both GSOMA variants.

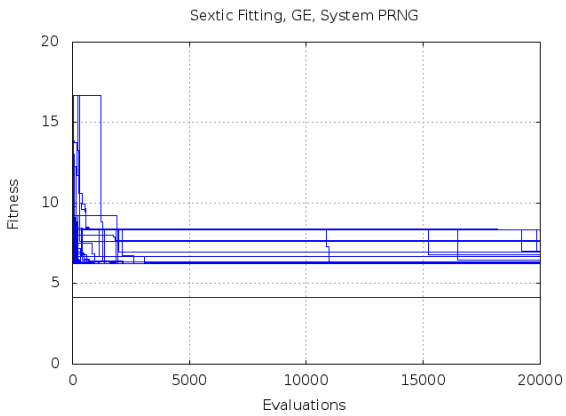
11.2 Experiment Results

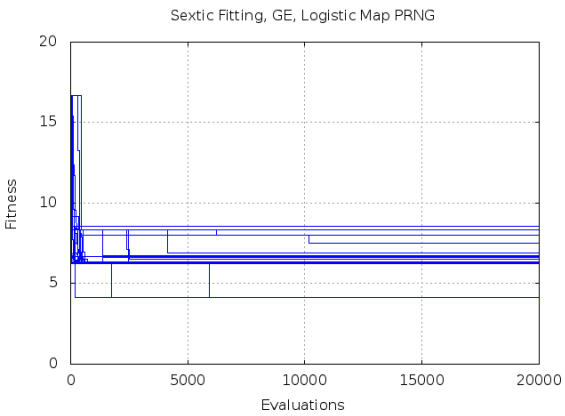
This chapter contains graphs that visualize the experiment results in different ways. The result format is the same as with the first experiment. The first type of diagrams show evolution progress as it tries to approach the null fitness value. Please refer to the Experiment Results section of the first experiment for detailed description.

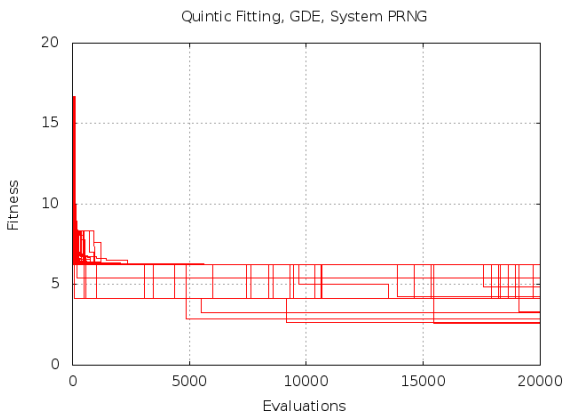
The second type of diagram shows the overall performance of each evolutionary technique depending of the PRNG used. There is a series for each evolution technique measured (each of them with both PRNGs) on the horizontal axis. The vertical axis shows the average fitness values of the best solution of each iteration that have been created using 20,000 fitness function evaluation at maximum.

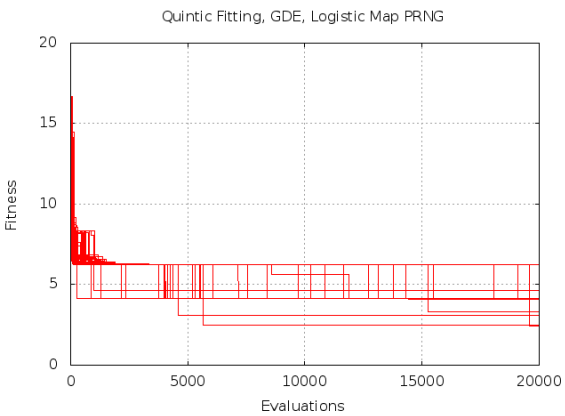

Quintic Fitting, GE, System PRNG
Fitness
Evaluations
0
5
10
15
20
0
5000
10000
15000
20000

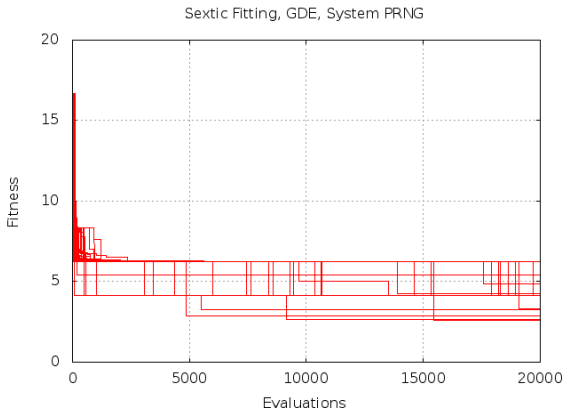

Quintic Fitting, GE, Logistic Map PRNG
Fitness
Evaluations
0
5
10
15
20
0
5000
10000
15000
20000

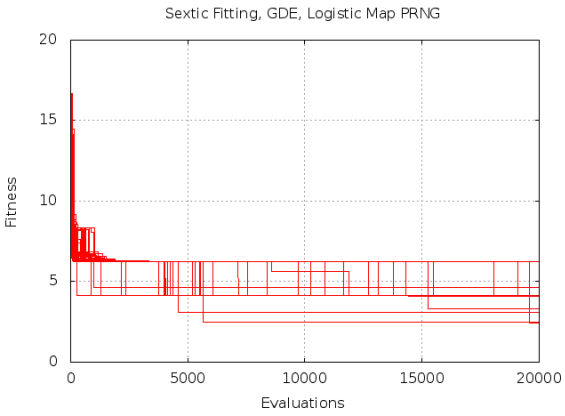

Sextic Fitting, GE, System PRNG
Fitness
Evaluations
0
5
10
15
20
0
5000
10000
15000
20000

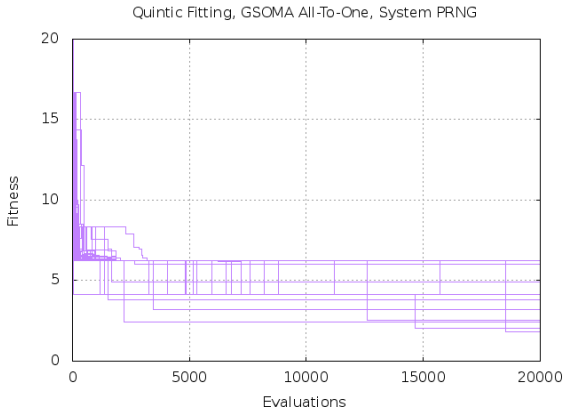

Sextic Fitting, GE, Logistic Map PRNG
Fitness
Evaluations
0
5
10
15
20
0
5000
10000
15000
20000

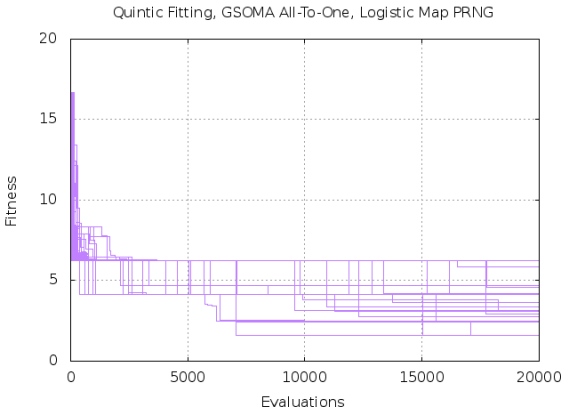

Quintic Fitting, GDE, System PRNG
Fitness
Evaluations
0
5
10
15
20
0
5000
10000
15000
20000

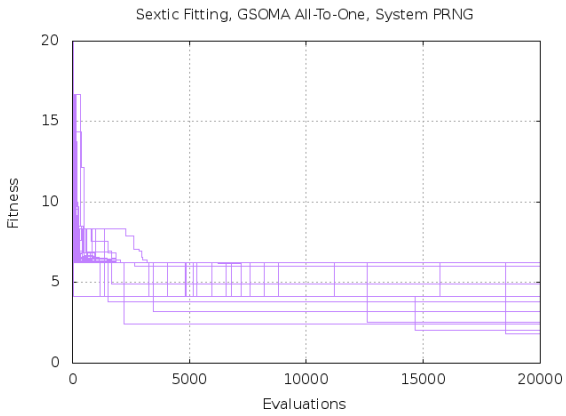

Quintic Fitting, GDE, Logistic Map PRNG
Fitness
Evaluations
0
5
10
15
20
0
5000
10000
15000
20000

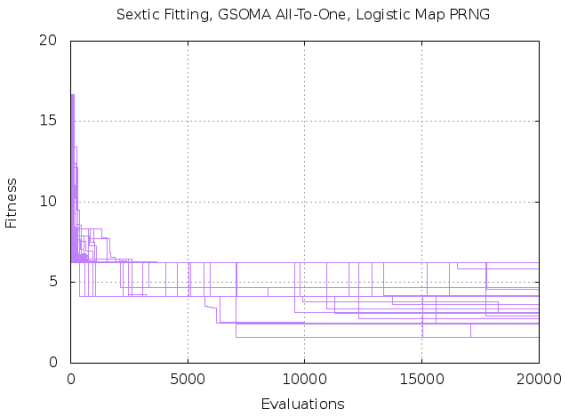

Sextic Fitting, GDE, System PRNG
Fitness
Evaluations
0
5
10
15
20
0
5000
10000
15000
20000

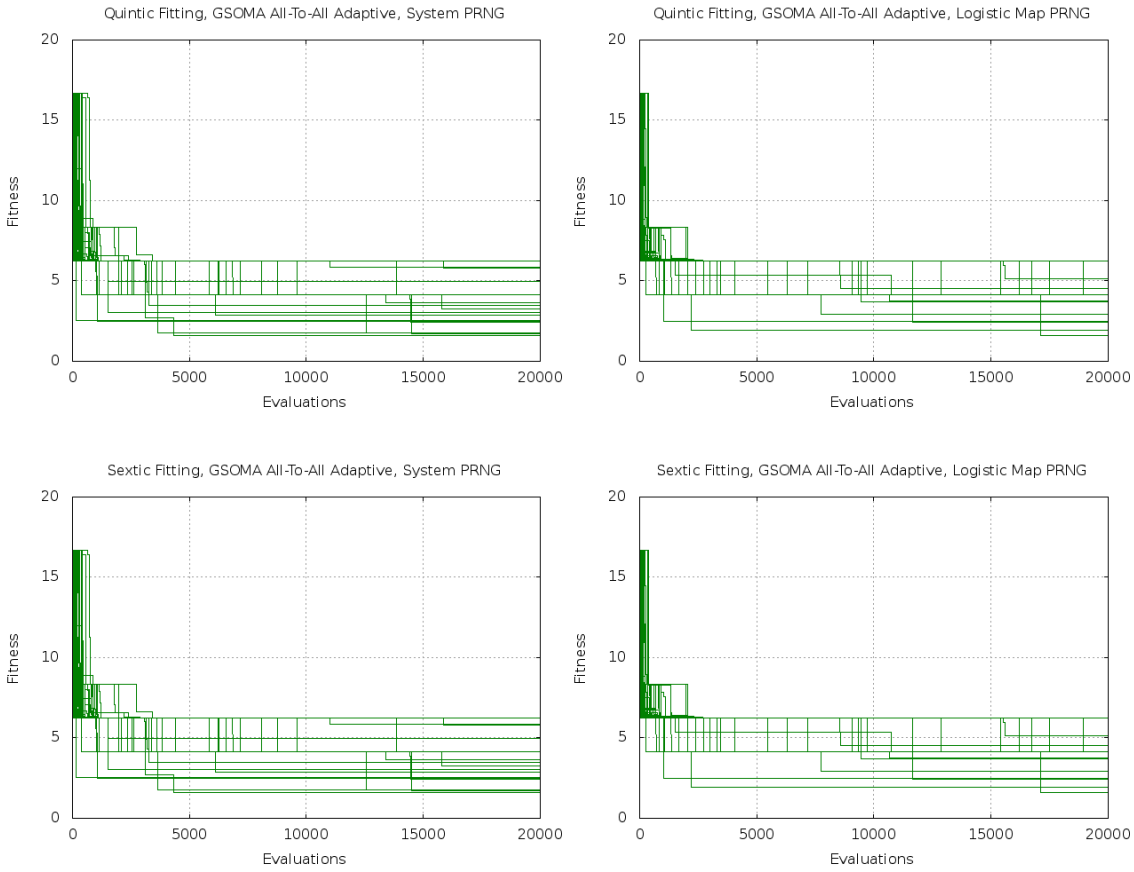

Sextic Fitting, GDE, Logistic Map PRNG
Fitness
Evaluations
0
5
10
15
20
0
5000
10000
15000
20000

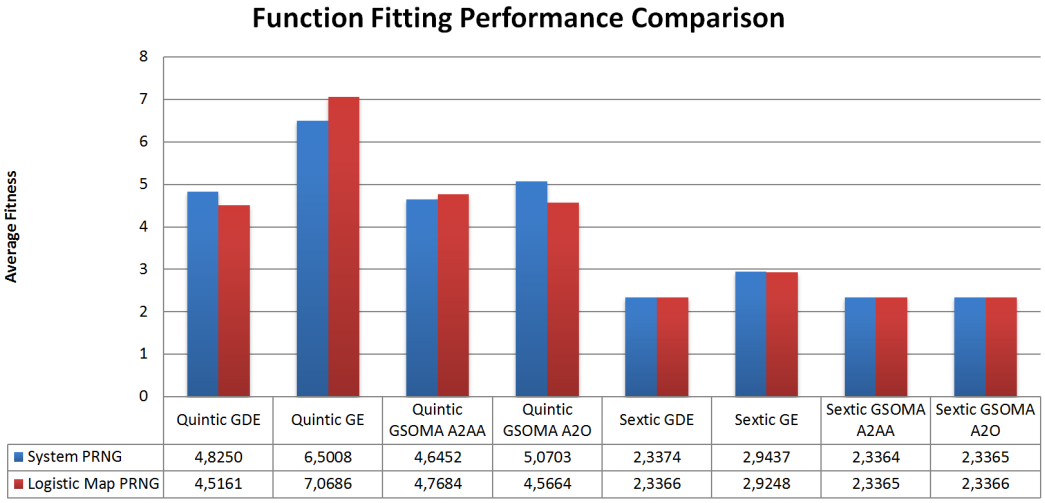

Quintic Fitting, GSOMA All-To-One, System PRNG
Fitness
Evaluations
0
5
10
15
20
0
5000
10000
15000
20000


Quintic Fitting, GSOMA All-To-One, Logistic Map PRNG
Fitness
Evaluations
0
5
10
15
20
0
5000
10000
15000
20000


Sextic Fitting, GSOMA All-To-One, System PRNG
Fitness
Evaluations
0
5
10
15
20
0
5000
10000
15000
20000


Sextic Fitting, GSOMA All-To-One, Logistic Map PRNG
Fitness
Evaluations
0
5
10
15
20
0
5000
10000
15000
20000





	Quintic GDE	Quintic GE	Quintic GSOMA A2AA	Quintic GSOMA A2O	Sextic GDE	Sextic GE	Sextic GSOMA A2AA	Sextic GSOMA A2O
System PRNG	4,8250	6,5008	4,6452	5,0703	2,3374	2,9437	2,3364	2,3365
Logistic Map PRNG	4,5161	7,0686	4,7684	4,5664	2,3366	2,9248	2,3365	2,3366

Figure 32: Function fitting performance comparison

	Sextic Fitting	Quintic Fitting
Avg. fitness using system PRNG	2.4885	5.2603
Avg. fitness using logistic map PRNG	2.4836	5.2299

Table 10: Functions comparison

11.3 Experiment Conclusion

Based on the presented results, the chaotic PRNG based on logistic map proved itself suitable for use with GE, GDE and both GSOMA variants. As it turned out, neither of the techniques is sensitive to the non-uniform numbers distribution of the logistic map PRNG. The difference in performance using either of generators is within a statistical error. It is surprising such simple dynamic system can produce random numbers that allows evolutionary techniques to run with almost unaffected performance. The uniformity of distribution is of course not the only attribute of quality and logistic map PRNG may have some other qualities that the system PRNG is missing, although it offers much better (i.e. more uniform) numbers distribution. It also does not seem that logistic map PRNG would work better with some evolutionary techniques that with the others. Results for GE, GDE and GSOMA are all very similar for both PRNG types.

12 User Manual

This section of the thesis aims on description of the implemented programs from the user's perspective. There is a subsection dedicated to each of the three programs. All of them are console applications and so interaction with them is done through a terminal window. Although each of them has some specifics, they share many common command line arguments for easier usage.

On Windows, it is enough to start programs by typing their name into a terminal window. On Mac OS and Linux the must be executed by the Mono runtime environment. See the following examples:

On Windows: ge.exe [arguments]

On OS X and Linux: mono ge.exe [arguments]

12.1 Common Arguments

The following tables (11, 12, 13) contain command line arguments that are common for all three programs.

/expr	A mathematical expression that represents the desired ideal solution. The evolution will try to find its approximation.
/from /to	The range on the horizontal axis on which the candidate solutions shall be approximated and evaluated.
/step	The sampling step size. Smaller numbers leads to more precise evaluation but also degrades evolution performance.
/grammar	A path to the file containing a grammar defined in the BNF format. The phenotypes will be generated accordingly to that grammar.

Table 11: Problem definition

/rng	The type of random number generator to use. There are two valid values: • **system** – The default .NET Framework generator (System.Random) • **logistic-map** – The chaotic generator using logistic map If the logistic-map option is selected, the **/logistic-map-a** parameter must also be defined.
/logistic-map-a	The parameter of the logistic map generator. Does only make sense if the logistic map-based generator is used. Otherwise, the program ignores it.

Table 12: Random Number Generator Settings

/itr	Determines how many times the whole evolution process shall be repeated. This is useful for experiments where more tries should be taken and their results approximated before making a conclusion.

Table 13: Other settings

Input Format

- Both "/" and "- -" characters can be used as a switch but not "-". Programs would not be able to distinguish between switches and negative numbers.
- Real numbers can be entered using either decimal dot or comma, independently on system settings. Both variant are always understood.

12.2 Grammar Evolution

The following table contains command line arguments that are specific for the grammar evolution program (ge.exe).

/gen	The number of generations to evolve. The evolution may stop earlier if an optimal solution is found.
/s	The selection strategy to use. There are three options: • **tournament** for tournament selection • **roulette** for roulette-wheel selection • **rank** for rank selection If the tournament selection had been chosen, the **/ts** paramerer must also be specified.
/ts	Determines how many individuals are chosen to conduct a tournament. Does only make sense it the tournament selection is chosen. Otherwise, it is ignored.
/pop	The number of individuals in each population.
/icl	The chromosome length, i.e. the number of genes that a genotype consists of.
/cr	Cross rate. Defines the probability that two individuals would cross. It must be in the $\langle 0; 1\rangle$ interval.
/e	The elitism level. Determines how many best individuals from a previous generation are to be preserved in a next generation.
/itr-best-only	This is a value-less parameter. If present, the application does only output the best solution of the last generation per iteration. That solution is also the best one of whole iteration, unless the elitism level is set to zero. The argument does only make sense if there is more than one iteration to be evolved.

Table 14: GE command line arguments

An Example

```
$ ge.exe /gen 100 /expr "Pow([x],6)-2*Pow([x],4)+Pow([x],2)" /gram
mar "SampleGrammar.bnf" /from -1 /to 1 /step 0.04 /s tournament /ts
4 /pop 2000
```

Approximates the $x^6 - 2x^4 + x^2$ function on the $\langle -1; 1 \rangle$ interval. The function is sampled using 0.04 step. The grammar defined in the "SampleGrammar.bnf" file will be used along with tournament selection. Five randomly chosen individuals will conduct tournaments and there will be 2000 individuals in each population. The evolution will run for 100 generations unless an optimal solution is found prior to the 100th generation.

12.3 Grammar Differential Evolution

The following table contains command line arguments that are specific for the grammar differential evolution program (gde.exe).

/gen	The number of generations to evolve. The evolution may stop earlier if an optimal solution is found.
/pop	The number of individuals in each population.
/chromosome-len	The chromosome length, i.e. the number of genes that a genotype consists of.
/cr	Cross rate. Defines the probability that two individuals would cross. It must be in the $\langle 0; 1 \rangle$ interval.
/f	The mutation constant. Must be within the $\langle 0; 2 \rangle$ interval.

Table 15: GDE command line arguments

An Example

```
$ gde.exe /gen 100 /expr "Pow([x],6)-2*Pow([x],4)+Pow([x],2)" /gram
mar "SampleGrammar.bnf" /from -1 /to 1 /step 0.04 /cr 0,9 /f 1.75
/pop 500 /chromosome-len 100 /itr 10
```

Approximates the $x^6 - 2x^4 + x^2$ function on the $\langle -1; 1 \rangle$ interval. The function is sampled using 0.04 step. The grammar defined in the "SampleGrammar.bnf" file will be used. There will be a 90% chance of crossing of two individuals. A mutation factor of 1.75 will also be used. The evolution will run for 100 generations unless an optimal solution is found prior to the 100th generation. The whole evolution will be repeated in 10 iterations.

12.4 Grammar SOMA

The following table contains command line arguments that are specific for the grammar SOMA program (gsoma.exe). The GSOMA implementation uses two terminating conditions:

- When the fitness divergence fall to or below the **/min-div** value (refer to the table for explanation).
- An ideal solution is found.

/migrations	The number of migrations to run unless an optimal solution is found earlier. In it an equivalent to the **/gen** parameter used to defined the number of generations with other programs.
/variant	The SOMA variant to use. There are two self-describing options available: • all-to-one • all-to-all-adaptive
/perturbation	The perturbation probability. Must be a number from within the $\langle 0; 1\rangle$ interval. When set to 0.3, e.g., there is a 30% probability that a dimension gets blocked.
/path-len	The length of a migration path relative to the distance of an individual from its target.
/mig-step	The length of a single migration step relative to the path length defined by **/mig-step**. It should not be a multiple of path length.
/pop	The number of individuals in each population.
/min-div	The terminating parameter. If the difference between fitness values of the best and the worst solutions are lesser or equal to the **/min-div** parameter, the evolution stops. If a negative value is set, the terminating condition is never satisfied. However, even in that case the evolution may stop before the defined number of migrations is done, if an optimal solution is found sooner.

Table 16: SOMA Command line arguments

12.5 Output Format

All three programs output the data in the same format. Since machine processing of the results is expected, the data are outputted in a very simple textual format, where "`\t`" stands for the tab character:

`[Iteration]\t[Generation]\t[Fitness Value]\t[Phenotype]\t[Fitness Function Evaluations]`

- **Iteration** is a one-based number of iteration. Iteration represents the whole evolution process.
- **Generation** represents the one-based number of generation within the iteration. In case of GSOMA, the number of migration cycle is outputted instead.
- **Fitness Value** is obvious. Fitness of the best individual from the generation is shown. It is always formatted using invariant culture, thus decimal dot, not comma.
- **Phenotype** is also obvious. Same as in the previous case, the phenotype of the best individual is used.

- **Fitness Function Evaluations** is a number representing how many times the fitness function had been evaluated until that time.

Saving Output to File

The output can be easily saved into a file by redirecting the output using standard redirection feature available on all supported platforms (Windows, Mac OS and Linux) using the "$>$" operator:

```
$ gde.exe /gen 100 /expr "Pow([x],6)-2*Pow([x],4)+Pow([x],2)" /gram
mar "SampleGrammar.bnf" /from -1 /to 1 /step 0.04 /cr 0,9 /f 1.75
/pop 500 /chromosome-len 100 /itr 10 > myfile.txt
```

The example above would store everything that the gde program outputs to console into a file instead. Please note that when output is redirected into a file, it is not printed into the console anymore and so it is not possible to see any progress there.

13 Conclusion

This third part of the thesis concludes achieved results and also suggests ways of further research of the topic.

13.1 Achieved Results

The first goal specified as "Introduction into state of art of grammatical evolution" has been carried out by elaborating on the topic in the first, theoretical part of the thesis. There have been principles of evolutionary algorithms described as well as three concrete evolutionary techniques – GE, DE and SOMA. Since neither DE nor SOMA are capable of producing expressions using a CFG by itself, a principle that enables such functionality have also been described. Apart from evolutionary techniques, there have been another two topics covered in the theoretical part:

1. Random number generators
2. Deterministic chaos

Both topics have been included because of the second experiment that aimed on measuring suitability of the PRNG based on logistic map for the purpose of generating pseudo-random number for evolutionary techniques.

Another part of the thesis that addressed the second goal specified as "Create program of grammatical evolution" aims on description of implementation details of GE, GDE and GSOMA applications. There is also a part that describes the common codebase used by all programs.

The third goal, specified as "Test of grammatical evolution on selected problems" has been carried out by two experiments.

- The first experiment aimed on comparison of evolution performance depending on the technique used. All three evolutionary techniques (GE, GDE and GSOMA) that the thesis deals with have been tested and their performance compared mutually.
- Another experiment also involved all three evolutionary techniques but instead of mutual comparison of their performance a chaotic PRNG based on logistic map has been used with each of them. The performance was then measured between each evolutionary technique using the default system PRNG and the same technique using the logistic map PRNG.

The fourth part describes each of the applications that have been implemented from users' perspective. Since they are console applications, tables describing command line arguments are present along with examples.

The implementation is based on modular architecture that allows easy replacement or extension of various parts. It also adheres to common software engineering principles such as KISS, DRY or SoC. Subjectively, I believe that the implementation is of decent quality and can be used as a starting point for further work.

Although GE is well-known evolutionary technique and even GDE is a proven concept [11], I am not aware that there would have been any research done regarding grammar-enabled SOMA (referred to as GSOMA in the thesis). Neither am I aware of any research regarding usage of a chaotic PRNG based on logistic map with GE, DE or SOMA. This makes me believe that the thesis is innovative in these two aspects.

Personally, I believe that all goals specified in the assignment have been carried out successfully. I also believe that the topic has potential to be applied on optimization of real-world problems and is therefore worth of further research.

13.2 Further Research

Grammar evolution is a big topic that belongs into even bigger field of evolutionary algorithms. Since possibilities of further research have not been exhausted by this thesis, this last chapter suggests ways of potential further research.

- The thesis described experiments with GE in classic sense, using genetic algorithms. It also dealt with the possibility of replacing genetic algorithms by DE and SOMA. Since grammar evolution turned out to be performing better with either of them, it might be interesting to test other evolutionary techniques such as PSO, simulated annealing etc.
- Similarly, it might be interesting to plug-in other chaotic RNGs and see how they perform compared to logistic map that has been used in the thesis. This should be an easy task thanks to the modular architecture of the implemented solution.
- The solution could be extended to support instrumentation for gathering diagnostics data. Combined with some graphical interface (or at least some form of reports), that would bring an interesting insight into the evolution process. By revealing how the evolution looks like in various times of execution, this insight could enable further optimizations or reveal inappropriately set control parameters. Or, if nothing else, the possibility to visualize the processes could be useful for educational purposes.
- Although BNF can express any CFG, codes tend to be lengthy due to the simple syntax. Another parser capable of either parsing some of the BNF supersets or completely different language could be added.
- The implementation could be modified to take a better advantage of parallel processing which is a feature that is currently limited. There would be two possible ways how to accomplish it – either by making a classical distributed application or by taking advantage of CUDA [23] in order to run computations on a GPU. It would be even possible to combine both approaches.
- Some stall-recovery techniques could also be incorporated into GE, GDE and GSOMA implementations to help evolution processes escape from local extremes.
- There is a variant of DE referred to as meta-differential evolution which is essentially a DE extended in a way that it can evolve its own control parameters. This approach

could also be used with GE, GDE and GSOMA. Since good control parameters setting is crucial for evolution performance, mitigating the need of setting them might be an actual improvement, especially in applications where there is no one to set them up neither they are known in advance. I believe that a reusable component could be made, allowing simple integration with various applications, even by people unfamiliar with evolutionary techniques in general [3].

Tomáš Machala

14 References

[1] Lažanský, J., Mařík, V., Štěpánková, O. *Umělá Inteligence 1*, předmluva

[2] Kvasnička V., Pospíchal J., Tiňo P., *Evolučné algoritmy*, STU Bratislava, ISBN 85-246-2000, 2000

[3] Zelinka Ivan, *Evoluční výpočetní techniky - principy a aplikace*, BEN, Praha, 2008

[4] Koza J.R. 1998, *Genetic Programming*, MIT Press, ISBN 0-262-11189-6, 1998

[5] Koza J.R.,Bennet F.H., Andre D., Keane M., *Genetic Programming III*, Morgan Kaufnamm pub., ISBN 1-55860-543-6, 1999

[6] O'Neill, M., Ryan, C. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*, Kluwer Academic Publishers, 2003

[7] Beasley D., Bull D. R., Martin R. R. *An Overview of Genetic Algorithms, Part 1, Fundamentals*, University Computing, 1993

[8] Zelinka, I., Oplatková, Z., Šenkeřík, R. *Aplikace umělé inteligence*

[9] Zelinka Ivan, *Umělá inteligence v problémech globální optimalizace*, BEN, Praha, ISBN 80-7300-069-5, 2002

[10] Lampinen Jouni, Zelinka Ivan, *New Ideas in Optimization & Mechanical Engineering Design Optimization by Differential Evolution*, Volume 1, London: McGraw-Hill, 1999. 20 p. ISBN 007-709506-5

[11] O'Neill, M., Brabazon, A. *Grammatical Differential Evolution*, International Conference on Artificial Intelligence, ICAI 2006

[12] Zelinka, I.: *Analytic Programming by Means of Soma Algorithm*, ICICIS'02, First International Conference on Intelligent Computing and Information Systems, Egypt, Cairo, 2002

[13] Knuth, Donald Ervin. *Umění programování: 2. díl, Seminumerické algoritmy*, Brno: Computer Press, 2010, xiii, 762 p. ISBN 02-018-9684-2

[14] *RANDOM.ORG, True Random Number Service*, [online], 1998-2014 RANDOM.ORG. Available on: http://www.random.org/

[15] Kathleen T., Alligood Tim D., Sauer James A. *CHAOS: An Introduction to Dynamical Systems*, Yorke Springer, 2000, ISBN 978-0387946771

[16] Marsland S. (2011) *Machine Learning*, CRC Press, §4.1.1.

[17] IBM, *System/360 Scientific Subroutine Package Version II Programmer's Manual*, H20-0205-1, 1967, p. 54

[18] NIST Special Publication 800-22 *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. Available on: http://csrc.nist.gov/publications/nistpubs/800-22-rev1a/SP800-22rev1a.pdf

[19] M. Matsumoto, T. Nishimura, *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator*, ACM Transactions on Modeling and Computer Simulation (TOMACS) - Special issue on uniform random number generation, Vol. 8, 3–30, 1998

[20] L'Ecuyer, P., Simard, R. *TestU01: A C Library for Empirical Testing of Random Number Generators*, ACM Transactions on Mathematical Software, Vol. 33, 4, article 22, 2007

[21] Elert, G. *The Chaos Hypertextbook*^{TM} [online], 1995-2007 [quoted 2014-04-14]. Available on: http://hypertextbook.com/chaos/42.shtml

[22] *Mono Compatibility* [online]. Available on: http://www.mono-project.com/Compatibility

[23] *CUDAfy.NET* [online]. Available on: http://cudafy.codeplex.com